AN APPROACH TO AUTOMATING SYNTAX ERROR DETECTION RECOVERY, AND CORRECTION FOR LR(K) GRAMMARS

Gordon T. McGruther

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

An Approach to Automating Syntax Error Detection Recovery, and Correction for LR(k) Grammars

by

Gordon T. McGruther

Thesis Advisor: R. C. Bolles

June 1972

Approved for public release; distribution unlimited.

An Approach to Automating Syntax Error Detection, Recovery, and Correction for LR(k) Grammars

by

Gordon T. McGruther
Lieutenant Commander, United States Navy
B. S., Naval Postgraduate School, 1971

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1972

ABSTRACT

An automatic, language-independent syntax error detection, recovery, and correction system for LR(k) grammars is proposed. The requirement is made that the reverse of the grammar involved is also LR(k). The implications and justification for this requirement are discussed. Given that the grammar is both LR(k) and RL(k), forward and reverse parsers localize errors and define left and right error context providing a strong base from which error analysis may proceed. Possible deterministic and heuristic corrective actions to follow error analysis are presented. The definition and selection of keys from the set of terminal symbols for the grammar which enable the reverse parser to be engaged upon error detection are discussed.

A model of the proposed system, implemented in an XPL compiler for a large ALGOL-like grammar, is described and the results of test programs are exampled and discussed.

Possible extensions to the system are presented and areas requiring further analysis are defined.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

The research for this thesis was done on an IBM 360/67 system at the Naval Postgraduate School.

I wish to express my thanks to LT(jg) Robert C. Bolles for his advice, encouragement, and guidance during work on this thesis.

I. INTRODUCTION

Most compilers and compiler writing systems have some kind of error detection and recovery mechanisms built in. Most provide a degree of error analysis and indicate to the user an error type and an approximate position of the error in the input stream. Diagnostic messages range from a reference number to full statements of suspected cause followed by parse histories. The suspected error symbol may be flagged with a pointer or referenced by name or both. Some error analysis systems are even sophisticated enough to specify the error symbol exactly and state the correction necessary.

If an error can be located precisely and defined without ambiguity then it seems logical that an immediate correction should be made and the processing allowed to continue. In general, it would seem that the more exactly an error could be defined the more efficiently the user's and computer's resources would be utilized.

Research indicates that despite appreciable effort, attempts to design comprehensive error processing systems to accompany the increasingly popular mechanical compilers, translator writer systems (TWS), and compiler-compilers has not been very successful. The error processing systems that do exist range from extremely simple recovery-only schemes to fairly complex attempts at error correction.

It is proposed that an efficient automatic syntax error processing system for LR(k) grammars can be defined. The system will operate as a function of a grammar only, its parameters being defined by the grammar analyzer and the grammar parsing function.

The objectives of such a system would be (1) to detect as many syntax errors as possible. Recovery systems that simply delete code to some predefined symbol do not afford the programmer maximum exposure of his code to the analytical processes, (2) to detect errors as early as possible to enable a more tenable recovery/correction scheme. Perhaps one of the most unsettling errors are those diagnosted as "NO PRODUCTION APPLICABLE." This type of error is generally associated with the precedence parsers and is the case of symbols being pushed onto the stack after having been interpretted contextually correct locally. The error is discovered when a subsequent symbol requires a reduction of the symbol stack and the error symbol does not fit any production definiton, (3) to make as many viable corrections as possible so as to allow continuous scan for maximum error detection; only as a last resort delete code to affect recovery, (4) to avoid generating new syntax errors by either correcting the error or affecting a complete recovery. The inefficiency in correcting an error (or worse, recovering from one) only to alter the code so as to create another syntax error is evident, (5) to avoid passing errors into the parse stack. This condition gives rise to the difficulties of having to "undo" emitted code, and (6) to define errors as exactly and completely as possible if only to provide more meaningful diagnostics should the error correction attempt fail.

The error correcting system will be defined to operate in an XPL compiler for LR(k) grammars whose reverse is also LR(k) and will be capable of correcting detectable error sequences of n symbols where n would be fixed when the compiler was constructed. For grammars meeting this restriction, forward and reverse LR(k) parsers can be defined and will be employed to localize errors and define error context.

The left context of an error is defined by normal LR(k) parsing of the input stream. The right context is defineable by employing the finite state machine representation of an LR(k) parser. Key symbols that uniquely define states in the FSM are selected from the set of terminal symbols for the grammar. When an error is detected, the next n symbols are ignored and the input code following the error sequence is scanned for a key symbol. When a key is located, the reverse parser is engaged to parse back to the error sequence. The right context thus defined, coupled with the left context provided by the forward parser forms a base from which error analysis may commence. Error corrections are defined by generating symbol strings of length n and comparing them with the error sequence.

The effectiveness of the system will be demonstrated by implementing the procedure for a non-trivial ALGOL-like language. The system was restricted from accessing the LR(k) parse stack. Though broad classes of errors are correctable, this restriction defined a small set of errors that is not easily corrected. For the event that the error could not be corrected deterministically, the error analyzer was defined to always heuristically select a symbol for insertion or replacement as an attempted correction. In this situation, the analyzer would continue to manipulate the symbol sequence between the forward parser and the key in an attempt to achieve correct syntax but there were cases where the resulting correction became unrealistic. Hence, it was necessary to place a restriction on the number of heuristic attempts that would be made to correct the error. The process was aborted if a complete correction was not affected in this many attempts, code was delected through the key, and forward parsing was restarted at the symbol following the key.

II. CURRENT SYSTEMS

As early as 1963 the need for automatic error analysis and correction systems to be part of syntax directed compilers was recognized. Efforts toward the accomplishment of this goal resulted in the design of systems with capabilities ranging from simple recovery to fairly complex recovery and correction. A sample of the spectrum may be found in considering briefly the works of Irons, McKeeman, Leinius, LaFrance, and Rich.

A. IRONS

Irons [5] designed a parse algorithm which was guaranteed to manipulate an input stream until it was syntactically correct for some defined grammar. Briefly, the mechanism involved carrying out all possible parses simultaneously. An error condition was defined when none of the current parses could continue. Error recovery and correction involved discarding the input stream from the error symbol until a symbol was found that would be syntactically correct for one of the existing parses. A string of symbols (including the null string) that would permit the selected parse to continue was then generated and inserted at the error point. Irons claimed the algorithm to be "relatively" efficient in terms of space and time requirements. However, it is conjectured that the algorithm would not be competitive in terms of space and time requirements if it was used on a larger grammar for a user-oriented language.

The algorithm accomplishes error correction but at a rather primitive level as it operates on the very simple mechanism of deleting

code rather than making any attempt to analyze the error relative to its total environment. No attempt is made to ascertain the extent of the error or its total local context. For example, if a missing punctuation symbol following a statement constituted the error, then it is highly probable that a correct following statement would be deleted in the search for the punctuation mark. Automatic wholesale code deletion such as this is a fairly severe price to pay for error correction, particularly when program logic may be destroyed.

B. McKEEMAN

In Reference 10, McKeeman examples the simple extreme. When an error condition occurs, the input stream is scanned for an obvious "stop" symbol for the language; the semicolon was used in the reference. The interim code, including the error condition, is deleted and parsing is re-initialized at the stop symbol.

The advantages to such a system are obvious--it is easily and efficiently implemented, it is fast, and it does not create any new syntax errors. However, as there is no attempt to correct an error, there is no possibility of executing. Additionally, the programmer also loses the opportunity to have all of his code scanned for syntactic continuity.

Example: IF...e_1...THEN IF...e_2...THEN...;

Error e_2 will not be found in the process of deleting code between error e_1 and the semicolon.

C. LEINIUS

Working with the LR(k) grammars, Leinius' parser constructor defines a set of right context symbols to be used for error recovery

for each partial parse existing in each state of the parser [9]. Locating a member of the set in the input stream allows the completion of a partial parse and the resultant reduction to be made. When an error symbol is read, a choice of recovery procedures is offered. The symbol string may be immediately scanned for one of the currently applicable right context symbols or the stack may be searched to determine if the symbol just read is a right context symbol for some partial parse existing deeper in the stack. If the stack search fails then a decision must be made as to the state in which scanning should commence to locate a right context symbol. This system is a more refined attempt at error recovery as the right context symbol offers a more local choice than simply scanning ahead to a stop symbol. But the system closely parallels Irons' in that it is also possible that wholesale deletions can take place while scanning for a required symbol. More important, however, more syntax errors can be generated.

Example: (X + e_1 (X + X))

The second left parenthesis will be deleted while scanning to the right looking for the required "X" with which to replace the error e_1 and that deletion will obviously create an additional error when the parser attempts to read the second right parenthesis at the end of the string.

D. LaFRANCE

LaFrance's error correction system employs groups of Floyd productions redefining a BNF language with necessary error productions build into the groups [8]. The error correction mechanism is based essentially on pattern matching. For errors involving unique productions, that is productions that require no context check, the symbol at the top

of the stack and/or the next input symbol are manipulated in accordance with an ordered set of transposition, insertion, and deletion rules. Otherwise, the applicable productions are expanded to three symbols ahead. These triples are then compared against the next four symbols from the input stream to find a match in a set of twenty patterns which defines a correcting modification to the input stream. If no match is found, the input stream is scanned until a symbol is located which will permit completion of a partial parse and control is then passed to the appropriate group and processing continues.

E. RICH

Rich [11] performed some preliminary work on an error correction system for mixed strategy parsers based on a scheme suggested by Gries [3]. It involves using legal triples to correct an error. A legal triple is an ordered, syntactically correct set of three terminal symbols for the grammar. The triples would be applied to the symbol prior to the error and the error symbol or the former and the symbol following the error for errors restricted to single symbols. In this manner a required deletion, replacement, insertion, or transposition would be defined.

Rich anticipated that error correction attempts would have to be limited and that such a system would require provisions to facilitate recovery from an error correction that was found to be wrong. This would entail saving all parsing information at the point of the error, perhaps in the form of a temporary parse stack operating locally in parallel with the main stack. More important, provisions for a means of cancelling any code emitted during an aborted error correction could

be required. Rich suggested that if a correction could not be applied then a unit of code (e.g., <STATEMENT>) would be deleted and a pseudo statement (e.g., a diagnostic message) substituted.

III. PROPOSED SYSTEM

The basic mechanics of the system were initially conceptualized as involving analysis of the input string following a syntax error. This analysis coupled with that which had preceded the error would provide a more cohesive context in which to analyze the error thus enhance error localization and definition and increase the probability of selecting the most applicable correction. Error analysis in this environment would be more definitive than schemes involving matching patterns of terminal symbol strings or extrapolating possible inputs from the analysis available prior to the error.

A. LR(k) GRAMMARS

The LR(k) grammars were selected as the class to which the system would apply as they and LR(k) parsing enjoy several advantages over simple and mixed strategy precedence (MSP) techniques: (1) the class of LR(k) grammars includes the precedence grammars, (2) the LR(k) parse stack provides an accessible and complete parse history to any point during processing of the object string. This deterministic context should permit more confident error analysis, and (3) all syntax errors are detected in read or lookahead states in the form of "ILLEGAL SYMBOL PAIRS," thus, the LR(k) parse stack is syntax error free.

LR(k) parsers may be represented by a characteristic finite state machine (CFSM) [2] which consists of two essential active states--read and lookahead. The lookahead states are required to resolve stacking/reduction decisions; that is, the next k symbols in the input stream define sufficient context to resolve the local conflict. Associated with

each state in the FSM is a unique accessing symbol. The accessing symbol is the terminal or nonterminal symbol from the grammar that has caused the recognizer to enter that state. In Figure 1, the nonterminal symbol <Block Body> is the result of a reduction made to a portion of the symbol stream already processed onto the parse stack and is the accessing symbol for read state 5. Read state 5 causes the next symbol in the code stream, s_1, to be read. If s_1 is the symbol END then a transition is made to reduce state 8, if s_1 is a semicolon then a transition is made to read state 36. These two symbols then become accessing symbols for their respective transition states. Similarly, the symbols BEGIN, END, ... WRITEON are accessing symbols for their respective states following read state 36. The terminal symbols that are state accessing symbols will play a significant role in the proposed system and will be discussed below.

The entire LR(k) parse stack is accessible and defines the complete parse history. As LR(k) parsing is deterministic, each new state is a unique transition from its predecessor. This deterministic trace through the FSM as a symbol string is processed, continuously confirms syntactic continuity as each state is entered. Therefore, it is generally not necessary to access the entire stack to determine left context for a specific symbol.

B. LR(k)/RL(k) GRAMMARS

To achieve error isolation and definition of error context, the stipulation was made that the grammar on which the error corrector would operate must be both LR(k) and RL(k). Then the construction of a LR(k) parser for the reverse of the grammar would enable bi-directional

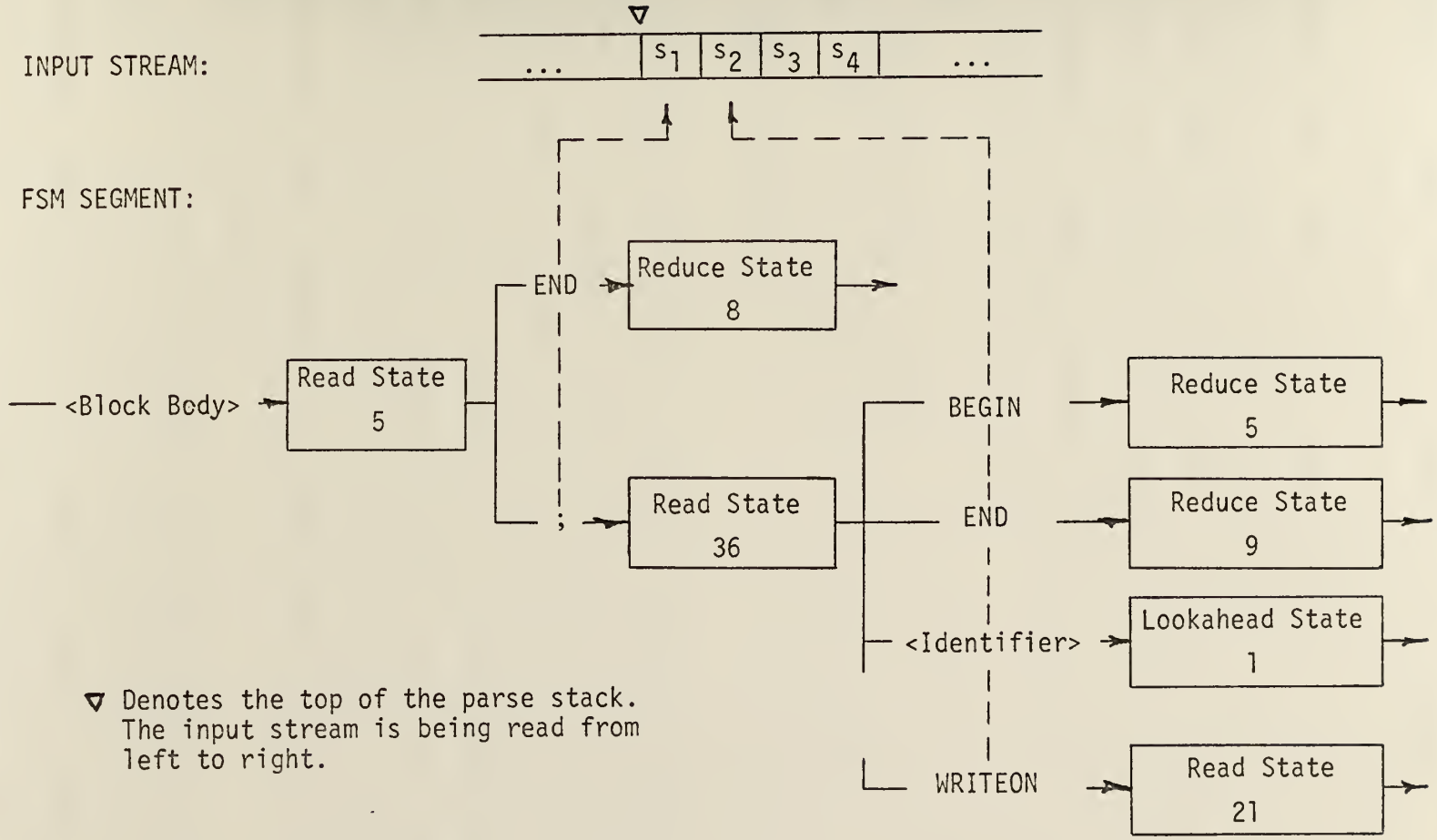
INPUT STREAM:
...
s_1
s_2
s_3
s_4
...
FSM SEGMENT:
<Block Body>
Read State 5
END
Reduce State 8
;
Read State 36
BEGIN
Reduce State 5
END
Reduce State 9
<Identifier>
Lookahead State 1
WRITEON
Read State 21
▽ Denotes the top of the parse stack. The input stream is being read from left to right.

Figure 1.

analysis of an error in that both forward and backwards parsers should recognize a given error in a sentence from the language.

It was fully appreciated that the above requirement was not insignificant. Knuth [7] discussed the LR(k)/RL(k) relation briefly by exampling a language for which a RL(k) grammar could be constructed but a LR(k) grammar could not, for any k. The specific problem that he exampled was encountered in the reverse situation in the grammar used in the model. Given the two ALGOL-E sentential forms:

FUNCTION <ID>(<ID>,<ID>,...,<ID>);

and

READ (<VAR>,<VAR>,...,<VAR>);

where <VAR> may be derived from <ID>, an input sequence:

READ (AAA,BBB,CCC);

is deterministic when read in a left to right manner because of the differentiating reserved word READ, but is not LR(k) when read right to left, because the parser cannot decide whether or not to reduce the identifier to <VAR> until the symbol READ has been recognized. This ambiguity was resolved in the model grammar used in this research by changing the read list delimiters from parentheses to vertical bars. (Two other similar changes to the grammar were required and will be described in Section IV.)

The cost of sacrificing minor user-oriented features should not necessarily preclude a language from more efficient processing techniques. Involved here is the sacrifice of minor symbology so as to permit automatic error processing of the grammar. Minor modifications of this same nature to specific grammars may enable the proposed system

to apply to a significant set of interesting languages. As the model grammar is not trivial, a valid example is provided.

C. REVERSE PARSING

Error definition and correction was approached from the point of view that they involved essentially the analysis of an error in its environment; and that the probability of not making a mistake during analysis was a function of the magnitude of the error environment considered. Thus, when an error is detected, it becomes necessary to read the input code that follows the error sequence and relate this right context to that on the left. In this manner the error would be localized. Pattern matching techniques, such as LaFrance's, accomplish this condition by projecting ahead all productions applicable at the point of error detection. This extension defines a set of all possible correct symbol patterns that may syntactically follow the last accepted symbol. LaFrance extrapolates all legal triples, thus, is able to correct most single and double symbol errors and, in some cases, triple symbol errors, particularly those involving reordering of the generated triples.

Except in the last case, at least one of the symbols in the generated triple was used to define the right context of the error. When one or more of the symbols in the triple were matched by symbols in the next four symbols from the input stream, corrections were based on the interpretation that the error extended from the symbol at which parsing halted to the start of the matching sequence.

It would be possible to also define right context by scanning the input stream to the end and allowing the reverse parser to parse from

right to left. When the reverse parser stopped due to an error the right context would be defined to that point. This can immediately be seen to be a very impractical method.

A means was needed to unambiguously engage reverse parsing at some intermediate symbol in the code stream beyond the error sequence. This would require the ability to uniquely define a parse state for that intermediate symbol. If a state could be so defined then, by the nature of LR(k) parsing, the parse history prior to that state could be inferred. Starting the reverse parser at an intermediate symbol would, in essence, simulate having parsed from the end of the code stream to that symbol.

If the symbol immediately following the error sequence could be determined and if this symbol was a FSM state accessing symbol, that is, it defined a unique state in the FSM, then an immediate transition to that state could be made. Associated with each read and lookahead state is a defined set of terminal symbols any of which is syntactically correct with respect to the accessing symbol for that state. When the transition was made, the reverse parser would be in a position to immediately reference the last symbol in the error sequence.

In this instance, only ordered pairs, vice triples, would be required for pattern matching as this is all that would be required to span the error sequence. The savings made by having to construct one less level of a generation tree are immediately apparent.

However, an immediate extension was suggested. If the symbol immediately following the error will define a unique reverse parse state then it may be possible to select any terminal symbol that so defines a state, find this symbol in the input stream, transfer to the appropriate state, and parse back to the error.

D. KEYS

The determination of symbols or keys that uniquely defined a reverse parser state was predicated on several requirements. Certain required attributes of a key were easily defined: the key should not be part of the error and it must appear in the code stream.

To ascertain that the keys were located outside of the error sequence required restricting the maximum length of the error sequence to n symbols. Then scanning for the keys would commence n symbols after the point of error detection.

The stipulation that the key must appear required that at least two symbols be designated keys. The first would be some symbol from the set of terminal symbols for the grammar and, to provide for the case where this symbol is not present in the balance of the input stream, the second symbol would be that used by the grammar to signify end-of-file.

Also, while keys could be located well beyond the error sequence, they should be located close enough so as to minimize the probability of encountering a second error while parsing back to the first.

A key that would specify a state in which reverse parsing could commence was only sufficient for reverse parsing. To provide for the case that the error was not correctable, it was also necessary that this key specify a state to which the forward parser could be transferred and restarted.

If the grammar is structured, as is the model grammar, then keys may be suggested by the delineators of the basic recursive forms. The basic form of an ALGOL-E sentence was quickly discerned as the terminal symbol BEGIN followed by any number of <Declaration Set>'s, each delineated by a period, followed by at least one <Statement>, with

semicolons separating multiple <Statement>'s, followed by the terminal symbol END. The period, semicolon, and END were considered as possible keys.

A grammar analyzer with which to define keys was not designed; however, a semi-mechanical analysis process was defined and applied to the model grammar.

After excluding the symbols <Identifier>, <Number>, and <String> from the model grammar, the intersection of terminal accessing symbols defining read states in the two parsers was found. The set contained only ".", ";", the set of arithmetic operators, "OR", "AND", "(", and ":" thereby eliminating END from the tentative list. Applying intuitive arguments, the set was further reduced.

All of the symbols, less the period and semicolon, were disqualified because they need not appear regularly in a code stream and they defined illogical potential deletion units. Long strings of code between the error sequence and the key would increase the probability of encountering a second error thereby causing the error correction attempt to be aborted and all code to the key to be deleted. An illogical deletion unit would be exampled by using the reserved word AND as a key and the attempt at error correction failed. Though it may be possible to delete code between two AND's and preserve syntactic continuity in the remaining code, intuitively, that deletion would violate the basic structure of the language.

The period and semicolon appeared to have both desired attributes. Judging from the language, both occur fairly regularly and more important, the strings of the code between either and an error are of manageable lengths.

Additionally, the left parenthesis and the add and subtract signs defined multiple states in which the reverse parser could be started. It would be a simple matter to scan for (say) a left parenthesis, but it would not be readily apparent in which state the reverse parser should be started to process code back to the error.

Though a simplistic approach, this general analysis of the grammar suggested several variants and extensions to the definition, employment, and effects of keys.

For example, the left parenthesis was found to be an accessing symbol for six read states in the reverse parser, three of which were independently unique. (The grammar analyzer employed to construct the parser was not designed to remove redundant states in the FSM, which it is possible to do.) As one of the prime objectives was to remain close to the error so as to avoid second errors as much as possible, it was seen that it could be significantly beneficial with respect to error correction capability to assign a symbol such as the left parenthesis as a key. The parenthesis is an often used symbol and its being designated a key would enable, in many cases, scanning and processing shorter strings of code. Resolution of the ambiguity created by the multiple states defined by the key could be accomplished by providing for variable path parsing via a system such as Irons'. That is, start the reverse parser in each state defined by the key and allow it to return to the error. It may be the case that an increased selection of possible error corrections may evolve, thereby enhancing the system's overall ability.

Secondly, only those symbols defining read states were considered for the model; however, it could be of benefit to not restrict key

selection to only that case. Through grammar analysis it may be possible and practical to define more valuable keys by considering those terminal symbols that define lookahead and reduce states in addition to read states.

In fact, a natural extension of the preceding discussion might be to consider only the symbol immediately following the maximum error sequence and allow variable path parsing back to the origin of the error. However, in the event that error correction failed, the problems associated with error recovery would remain to be resolved. It would be highly probable that the sequence of code between the error point and the key would not be a convenient string to delete. One possible solution would be to delete code to the first available key that did define a logical deletion unit.

Consideration of the above possibilities was doubly motivated. First by the objective to keep keys as close to the error as practical, and second, it was surprising to find a set of fifty terminal symbols so severely reduced when the subset of those symbols defining read states in both parsers was determined. It seems very likely that there may be interesting LR(k) grammars that would be excluded from the proposed system by restricting the definition of keys to those symbols that mutually defined only read states between the two parsers.

E. PROCEDURE

When an error is detected in either a read or lookahead state, the corrector procedure requires stepping over n symbols to insure that the key selected is not imbedded in the error string, scanning forward until a key is encountered, and engaging the reverse parser in the

state prescribed. The reverse parser is allowed to parse backwards until it either stops at the same point at which the forward parser stopped or is stopped due to encountering an error. If the length of the symbol string between the two parsers is greater than n then the restriction on error magnitude has been violated, code will be deleted to the key and the forward parser will be restarted at the symbol following the key. If the number of symbols between the two parsers is equal to k, 1<= k<=n, then symbol strings of length k are generated from the context of either parser and, via a set of pattern matching rules such as those defined by LaFrance, the generated strings are compared with the error string and either symbol deletion, insertion, replacement, and/or transposition will be defined. If k is equal to zero then the reverse parser has returned to the symbol recognized as an error by the forward parser. The error may be quickly resolved by intersecting the symbol sets associated with the two parse states thereby defining a replacement symbol. Or deletion may be defined by determining that both parsers would be satisfied by the symbols that follow the error relative to either parser.

In the case that the reverse parser is not in an error condition while reading the forward parser error symbol, an insertion symbol may be defined by intersecting the parse state symbol sets after stepping the reverse parser to its next read or lookahead state.

In the event that all deterministic error correction attempts fail, it may be advantageous to heuristically select a symbol from the forward parser symbol set to either replace or be inserted in front of the error symbol and restart forward parsing rather than automatically proceed with code deletion.

At the cost of the extra processing time required, a heuristic attempt to correct an error would serve two purposes. It may provide the necessary impetus to complete the correction or, even if the attempt failed, it should define to the programmer an approach to correction through the associated diagnostics.

Consider the case where the allowable error magnitude is one symbol and the error is actually the omission of two symbols. For example:

X := Y + __ IF A THEN...,

where the symbols "Z;" have been omitted. The forward parser will detect an error when it attempts to access the symbol IF and the reverse parser will detect an error accessing the plus sign. Neither parser may be satisfied by any deletion of adjacent errors, nor by the transposition of any symbol pairs. Also, the intersection of the symbol sets associated with each parser state will be empty, thus an insertion or replacement symbol will not be deterministically defined. A heuristic attempt to correct may be made at this point by selecting a symbol from the forward parser symbol set for insertion in front of the error symbol (the error symbol is the word IF for the forward parser.)

Obviously, by inspection, a choice is available. The selection would certainly include a number, another identifier, and a left parenthesis. Two of these three symbols would effectively reduce the remaining error to a single symbol and permit the deterministic processes to re-analyze the error.

If the left parenthesis was selected then the gains are not so obvious. On the next analysis iteration it is probable the deterministic attempts would again fail. Heuristically, however, another symbol would be inserted.

How symbols are selected from applicable sets is also variable. Whether they are selected as they are ordered in the set or in reverse order may be problematic. However, a means to avoid issuing a duplicate of the previous choice would probably be required.

In the manner described and within the confines of error restrictions, the proposed error corrector accomplishes error detection as early as possible and defines error processing such that the error is not promulgated to the stack. A strong deterministic attempt will be made to correct an error and failing that, a heuristic choice of correction will be applied.

Two other facilities would be required to support the proposed system: (1) an upper limit to the number of heuristically selected corrections that would be made for any one error must be specified. Only when this limit was reached would code be deleted, and (2) complete communications are maintained with the programmer to insure that, in the event error correction failed, the diagnostics would provide a complete history of corrector action helping to isolate, and perhaps allowing the user to quickly discern the true cause of the error.

The case that the key symbol had been missplaced and in itself constituted an error required consideration. No problem would arise if a key was located in the allowable error string as this string would not be considered when scanning for keys. If the key was erroneously placed beyond the error string then the error restrictions would be violated; however, the violation would not be detected until the code sequence between the key and the following key was processed. The corrector would not recognize an erroneous key in itself; hence,

correction procedures would be applied to both strings of code, that preceding the error key and that immediately following.

The possibility of defining symbol strings vice single symbols as keys to alleviate the problem of keys being in error was considered. Again, these considerations were also motivated by the desire to place the keys as close to the error as possible to preclude encountering second errors.

It may be possible to define ordered sets of terminal symbols such that their being located in the input stream would specify a unique start state for the reverse parser whose accessing symbol would be one of the elements of the set. For example, if the string <Operator> (<Identifier> uniquely defined a reverse parser state such that the accessing symbol was a left parenthesis, then the location of this string following an error may preclude the requirement to scan further for a semicolon or period. Thus, the possibility of encountering a second error while reverse parsing would be reduced.

The above concept of keying on symbol strings may be extendable to enable the forward parser to perceive or extrapolate symbol sets based on the state it was in when the error was recognized and the left context, that, if located in the code stream following the error, would define unique start states for the reverse parser. It may be possible to define a set or hierarchy of such strings through a complex analysis of the forward and reverse parser interface. Continuing the example above, for a given forward parser state there may be several contexts in which a left parenthesis may be taken such that each uniquely defines a reverse parser start state.

Not locating such strings following an error would not necessarily constitute a second error and would require that hierarchical sets such as these also include any "primary" keys defined for the grammar, such as the period and semicolon previously discussed. If the forward parser was currently parsing an <If Statement>, for example, and locating the reserved word THEN would enable engaging the reverse parser; not locating that key should not automatically constitute a second error. That particular key may be involved directly in the detected error sequence and scanning should continue, searching for the next defineable key in the key set for <If Statement>.

IV. IMPLEMENTATION

For the purpose of implementation of the error recovery system defined, considerations were restricted to those syntax errors involving only single symbols and transposition of symbol pairs. Extensions of the system to include errors of greater complexity and scope will be discussed at the conclusion.

A. COMPILER

A basic model of the proposed error correction system was implemented in an XPL compiler for ALGOL-E, a non-trivial ALGOL-like language (134 productions, 50 terminal symbols, 74 non-terminal symbols). A listing of the grammar is provided in the Appendix. The model is semmantics independent, its parameters being solely derived from the forward and reverse parsers, i.e., parse states and associated symbol sets.

The compiler was constructed from an existing ALGOL-E compiler employing MSP parsing [6] and an XPL skeleton compiler written by DeRemer [1] for his SLR(k) parser. Figure 2 shows some of the detail in the construction of the hybrid model compiler. Studies have shown that the SLR(k) parser constructor and the resulting parser to require significantly less space and time than the MSP parsers [2,4]. This was also found to be the case in this application. The SLR(k) parser for ALGOL-E required approximately 64 percent of the space required for the MSP parser for the same grammar. This was considered significant as the error correction technique to be implemented would require both a forward and a reverse parser.

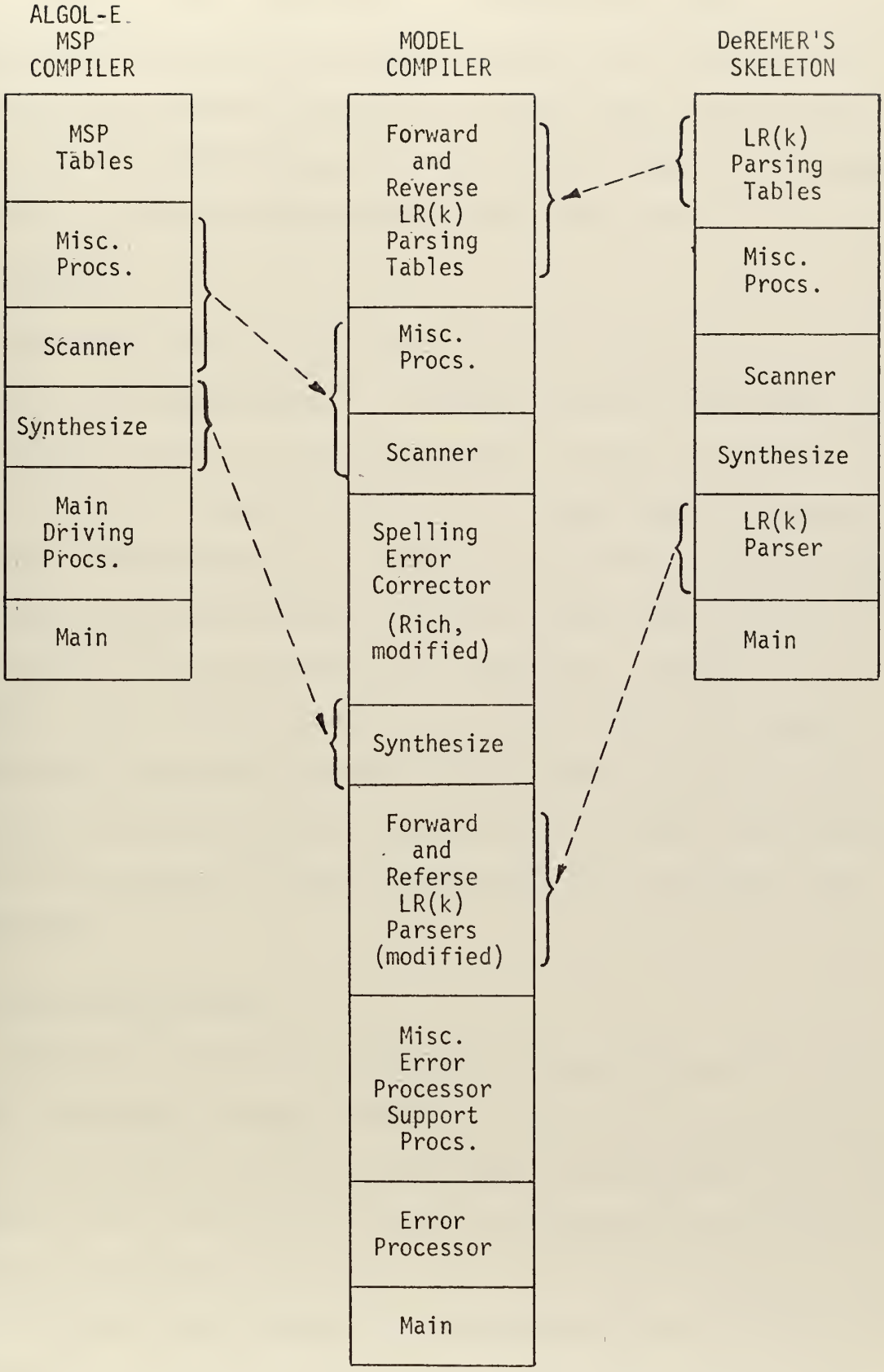
ALGOL-E MSP COMPILER
MSP Tables
Misc. Procs.
Scanner
Synthesize
Main Driving Procs.
Main
MODEL COMPILER
Forward and Reverse LR(k) Parsing Tables
Misc. Procs.
Scanner
Spelling Error Corrector (Rich, modified)
Synthesize
Forward and Referse LR(k) Parsers (modified)
Misc. Error Processor Support Procs.
Error Processor
Main
DeREMER'S SKELETON
LR(k) Parsing Tables
Misc. Procs.
Scanner
Synthesize
LR(k) Parser
Main

Figure 2

The SLR(k) parser constructor was defined and implemented by DeRemer. The gained efficiency of his system over other basic LR(k) parser constructors was achieved by constructing a LR(0) parser for the grammar then adding lookahead states only where they were needed. This approach resulted in faster construction and reduced parser size.

B. GRAMMAR

The ALGOL-E grammar [6] was found to be not SLR(1), as was also the case for the reverse grammar. The required changes to the grammar were essentially minor and did not detract from or enhance the language. It was necessary to change the delimiters in a read statement from parentheses to vertical bars and the ambiguity of the ALGOL assignment symbol, :=, was resolved by defining a new terminal symbol:<Setq>, <Setq> is transparent to the programmer as are <Identifier>, <Number>, and <String> and is similarly assigned in procedure SCAN of the compiler. Additionally, procedure calls were differentiated from function calls by requiring the reserved word CALL to precede the name of the subroutine. It was also necessary to delimit <Declaration Set> with periods vice semicolons.

C. SPELLING CORRECTIONS

Emperically, misspelled identifiers and reserved words form a significant percentage of errors; therefore, after appropriate modification, a spelling checking system was incorporated into the compiler [11]. An attempted error correction would fail if the reverse parser failed to return to the point of the input stream at which the forward parser was halted, hence, it was necessary to also enable spelling correction of misspelled reversed words in the reverse parser. Only reserved words

are pertinent to the reverse parser spelling checking procedure as it is concerned with only the syntax of identifiers, not the semantics, i.e., spelling. The spelling checking procedures incorporated were simplistic but demonstrative; only those errors involving one deleted or added character, one character in error, or two adjacent characters transposed were correctable. However, the complexity and sophistication are easily extended if one is willing to absorb the additional cost in terms of space and time.

D. PROCEDURE

The model consists of two primary procedures, ERROR_ANALYZER and REVERSE_PARSER (Reference Figure 3). CAN_DO_WITHOUT_TOP, FP_INSTRSCT_RP, and CHECK_CONTEXT_OF_TOP_AND_TOKEN are called from ERROR_ANALYZER to determine if a symbol is a member of an applicable symbol set or to determine the symbol in the intersection of the applicable symbol sets of the forward and reverse parsers respectively. The applicable symbol sets are those read and/or lookahead symbol sets for a particular forward or reverse parse state. Procedures TRANSPOSE, REPLACE, DELETE, and INSERT are called when a tentative error solution has been determined and the action implied by the procedure names is to be applied to the symbol at the top of the stack and/or the token symbol (next symbol to be read).

As in the case of spelling correction, the scope of errors was restricted to single symbol insertion or deletion, one symbol in error, or two adjacent symbols transposed.

Error analysis was restricted to only that symbol on the top of the stack and/or the token symbol. This restriction was imposed to

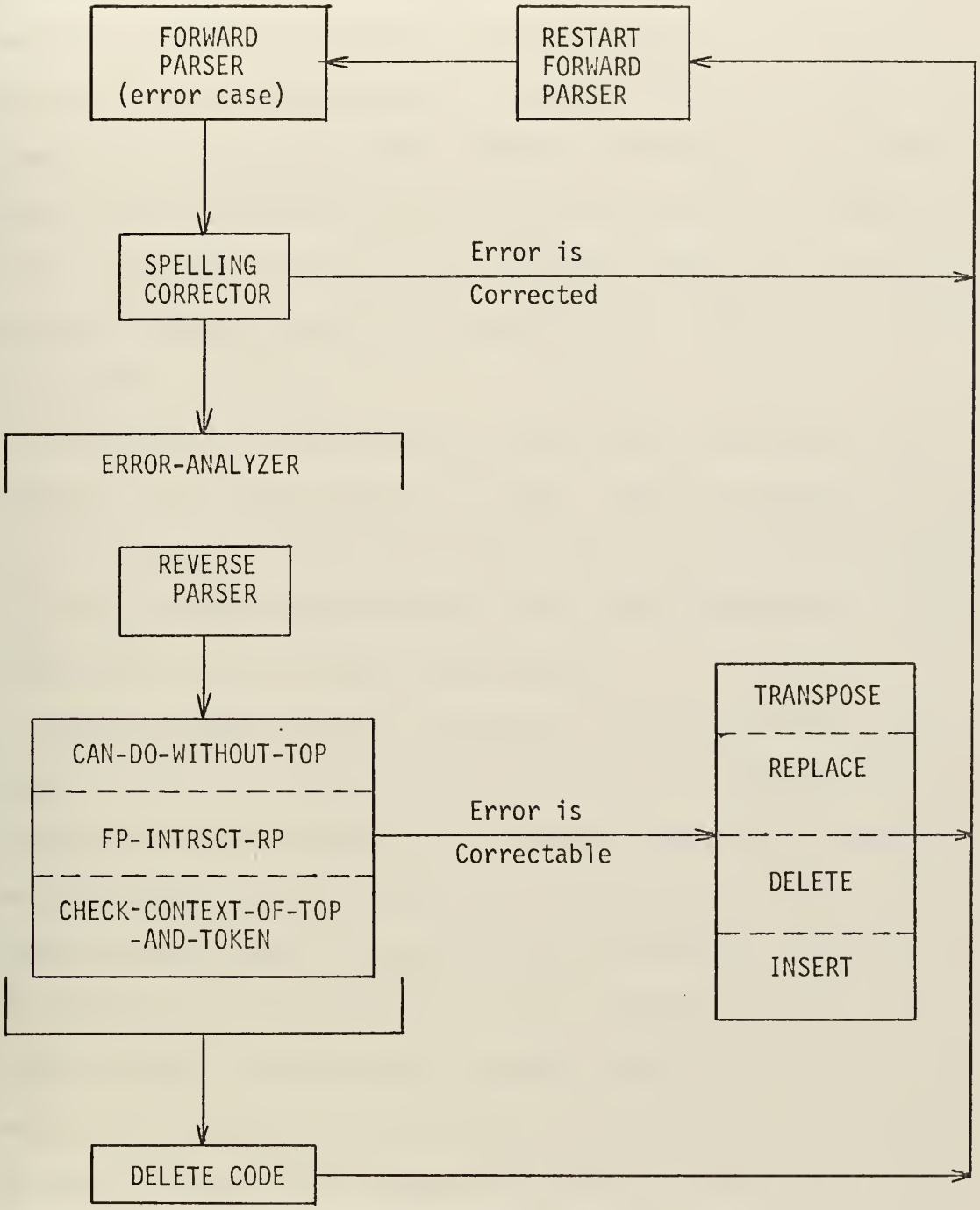
FORWARD
PARSER
(error case)
RESTART
FORWARD
PARSER
SPELLING
CORRECTOR
Error is
Corrected
ERROR-ANALYZER
REVERSE
PARSER
CAN-DO-WITHOUT-TOP
FP-INTRSCT-RP
CHECK-CONTEXT-OF-TOP
-AND-TOKEN
Error is
Correctable
TRANSPOSE
REPLACE
DELETE
INSERT
DELETE CODE

Figure 3

preclude having to delete code that may have been emitted with the possible reduction of the second symbol in the stack prior to detecting the error. Further, the heuristic choice was made to first test for the possibility of deleting the error symbol. This was to reduce the occurrences of having to define a <Number>, <Identifer>, or <String> should the case be that the error was caused by any one of those omissions. For example, if X:=Y++Z; was the input string then one of the operators would be deleted vice inserting either <Number> or <Identifier> or any other expression.

For purposes of implementation, the period and semicolon were defined as the primary keys for all cases. EOF was designated the terminal key. The period was used as the primary key when the syntax analyzer was parsing declarations (reference ALGOL-E (Modified) grammar listing) and semicolon was the primary key elsewhere.

When the forward parser is stopped by an error condition it is in either a read or a lookahead state and either the two top symbols on the stack or the top symbol and the lookahead symbol will constitute an illegal symbol pair. At this point, the history of the finite state machine for the grammar is known or may be determined directly from the current parse state and the set of read or lookahead symbols associated with that state. That is, given a symbol from the current applicable set, either the symbol will be stacked, indicating that the right part of some production is one symbol more complete, or the symbol just looked at will specify that the right part of a production has been completely read and a corresponding reduction will be made in the stack. The result of that reduction will in turn specify another symbol (a production left-part) toward completing the right part of some production entered further down in the stack.

If the error symbol cannot be corrected as a misspelling then the error analyzing mechanism is engaged. Symbols are read into a symbol stack while the input stream is scanned for a key. The reverse parser is initiated in the state specified for the key and, operating with its own state stack, processes the symbol stack in reverse until it is stopped by an error that it cannot resolve as a misspelling or it reaches the point in the code stream at which the forward parser stopped. For example, reference Figure 4.

Figure 4(a) depicts the configuration of the forward parsing stacks when an error (e) has been detected. The symbol e represents an error sequence of length n or less. If NEXT_SYMBOL(SP) is <Identifier> and is determined to be a misspelled reserved word then the correction is made immediately and parsing resumes; otherwise, the point of progress of the parse stack is marked (SAVE_SP, Fig 4(b)). The input stream is read to the key and the reverse parser is started in the read state for that key (R_STATE_STACK(RP)).

Figure 4(c) depicts the configuration of the stacks after the reverse parser has successfully parsed back to the error point and error analysis and correction begins. (Note: pointers SP and SAVE_SP have been interchanged for compiler execution considerations only.) When forward parsing resumes after error correction, symbols through the key are read from stack NEXT_SYMBOL. Only then does the parser return to reading the input stream. If the error cannot be resolved or the reverse parser is halted short of error e by additional errors then the code from the error to the key (NEXT_SYMBOL(SAVE_SP)) is deleted.

Figure 5 depicts various configurations the two parsers may be in when the reverse parser has stopped. In conditions 5(i) and 5(j) the

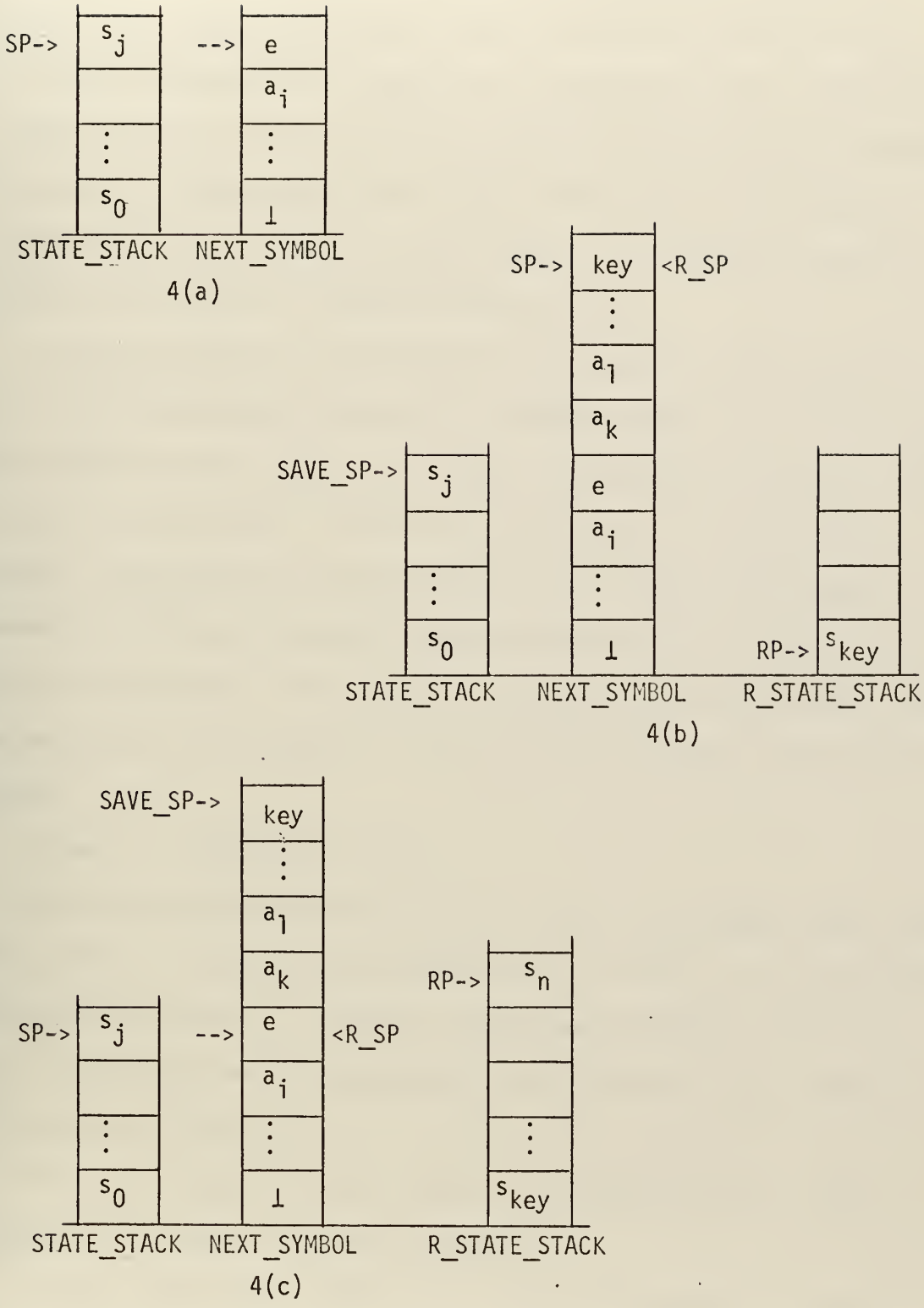
SP->
s_j
-->
e
a_i
s_0
⊥
STATE_STACK
NEXT_SYMBOL
4(a)
SP->
key
<R_SP
a_1
a_k
SAVE_SP->
s_j
e
a_i
s_0
⊥
RP->
s_{key}
STATE_STACK
NEXT_SYMBOL
R_STATE_STACK
4(b)
SAVE_SP->
key
a_1
a_k
RP->
s_n
SP->
s_j
-->
e
<R_SP
a_i
s_0
⊥
s_{key}
STATE_STACK
NEXT_SYMBOL
R_STATE_STACK
4(c)

Figure 4

errors are defined to be too far apart and symbols e_1 to key are deleted and forward processing is re-initialized at the semicolon. The conditions depicted in Figures 5(a) through 5(h) fall within the scope-of-error restrictions imposed and error analysis may be performed. Note that in configurations 5(a), 5(b), 5(e), and 5(f) the reverse parser may or may not be in an error state, i.e., symbol e_1 may be syntactically correct as the left context of a_1 .

For configurations 5(a) through 5(h), symbols a_i , e_1 , e_2 , and a_j are checked against the read or lookahead symbol sets for the forward and reverse parser states so as to make an appropriate deletion, insertion, or transposition. If the error cannot be so resolved then a symbol is heuristically selected from the applicable forward parser symbol set without reference to the reverse parser and inserted in front of the error symbol. This heuristic approach may be applied four times before code will be deleted. Control is then returned to the forward parser.

Example 1: Configuration 5(a)

Both the forward and reverse parsers are in read states after reading symbol e_1 . Let the forward parser be in state f_k and the reverse parser be in state r_k Let fss_k be the set of symbols associated with the forward parser read state f_k and similarly, rss_k represents the symbol set for r_k .

If a_j is a member of fss_k and a_i is a member of rss_k then delete e_1 and continue normal processing.

If the reverse parser (RP) is not in an error condition then step RP to its next read or lookahead state (r_{k+1}). If the intersection of fss_k and rss_{k+1} is empty then replace e_1 with the intersection of

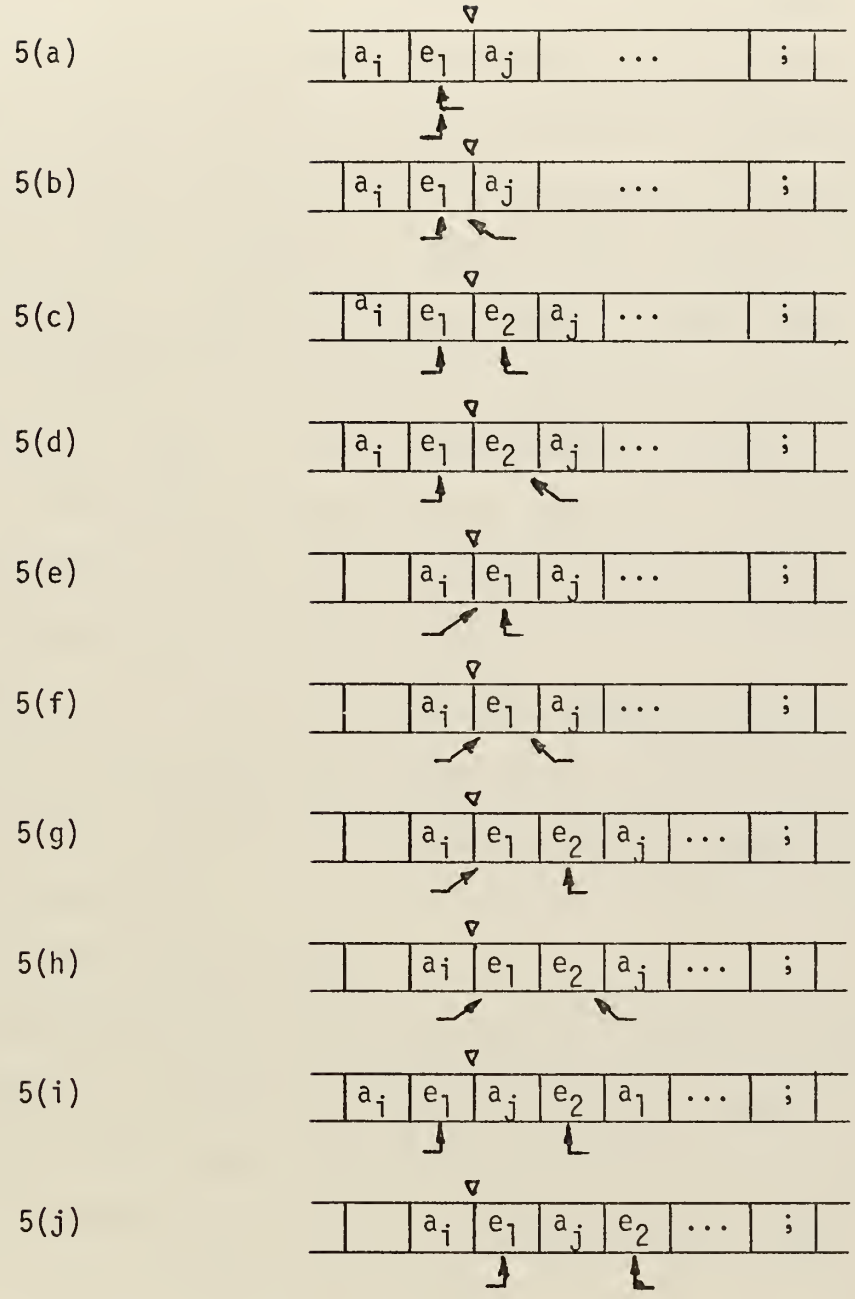

Figure 5

fss_k and rss_k (if this intersection is also empty then replace e_1 with a symbol from fss_k) and continue processing; otherwise, insert the intersectionof fss_k and rss_{k+1} in front of e_1 and continue. (Note: That the reverse parser may not be in an error condition when it reads the symbol causing the error for the forward parser is very pertinent to the error analysis process. If it is the case that it is not in error then the initial assumption is that a symbol is missing in front of the error symbol. With that assumption made, a symbol that is syntactically correct for both parsers is required for insertion in front of the error symbol. This is accomplished by stepping the reverse parser to its next read or lookahead state, which ever occurs first. The insertion symbol is then taken from the intersection of the symbol sets associated with the two parse states.)

Otherwise, (RP is in an error condition), replace e_1 with the intersection of the FP and RP read state symbol sets if that intersection is not empty (if that intersection is empty then insert a symbol from fss_k) and continue processing.

Example 2: Configuration 5(d)

Both of the parsers are in error conditions, the forward parser (FP) is in read state f_k and RP is in lookahead state r_k . Again, let fss_k and rss_k be the symbol sets for the respective parse states.

If e_2 is a member of fss_k and e_1 is a member of rss_k then transpose e_1 and e_2 and continue processing.

If the intersection of the two symbol sets is not empty then replace e_1 with a symbol from fss_k ., delete e_2, and continue.

If e_2 is a member of fss_k then delete e_1 and continue.

Otherwise (attempt the last resort), replace e_1 with a symbol from fss_k and continue processing.

E. RESULTS

Figure 6 examples some of the results of the error correcting system described. Generally, the system recognized a broad class of single symbol errors of insertion and omission and double symbol transposition errors. However, there was one small, well-defined class of error that though recognized, could not be corrected while retaining the imposed restriction of not modifying the parse stack below the top symbol to achieve an error solution.

For those constructs in which a statement was started with a reserved word followed by an identifier, the omission of the reserved word was not detected until the symbol following the identifier was read as it is syntactically correct for statements to also start with an identifier. In this instance, the true error point was two symbols from the top of the parse stack when the error condition was recognized.

In the case where the reserved word was not omitted but merely grossly misspelled such that the symbol was interpretted as an identifier, the error condition arose when the following identifier was read. In this instance the true error point was one down from the top of the stack.

For both situations, the omission and misspelling of the reserved word, by the time the error was discovered, the identifier following the error had already been reduced and associated code emitted.

For the class of error conditions that was processed correctly, most conditions were corrected in a logical manner; logical in the sense

```
A L G O L - E   C O M P I L A T I O N   - - -   L R ( K )   P A R S I N G

TODAY IS JUN 2, 1972.  CLOCK TIME = 3:38:50.64.

CARD |BL| SYL |
   1 |  0|   0 |LOCAL ALPHA, BRAVO, CHARLIE.
   2 |  0|   0 |ARAY DELTA <0 D 5>.

** CORRECTION ** INSERTING "BEGIN" BETWEEN "EOF" AND "LOCAL" ON LINE 1

** CORRECTION ** MISSPELLED RESERVE WORD "ARAY" REPLACED BY "ARRAY" ON LINE 1

   3 |  1|   1 |FUNCTION COMP (ECHO, FOXTROT)).

** CORRECTION ** REPLACING "D" BY "UNTIL" ON LINE 2

** CORRECTION ** REPLACING "UNTIL" BY ":" ON LINE 2
   4 |  2|  18 |      COMP ECHO + 5 * FIXTROT.

** CORRECTION ** DELETING ")" ON LINE 3
   5 |  2|  18 |PROCEDURE QUITTING

** CORRECTION ** DELETING "ECHO" ON LINE 4

** CORRECTION ** INSERTING "<SETQ>" BETWEEN "COMP" AND "+" ON LINE 4

** CORRECTION ** MISSPELLED IDENTIFIER "FIXTROT" REPLACED BY "FOXTROT" ON LINE 4
   6 |  1|  32 |      WRIT ("TH-TH-THAT'S ALL, FOLKS.").

** CORRECTION ** MISSPELLED RESERVE WORD "WRIT" REPLACED BY "WRITE" ON LINE 6
   7 |  1|  32 |READ BRAVO|;;

** CORRECTION ** INSERTING "." BETWEEN "QUITTING" AND "WRITE" ON LINE 6

** CORRECTION ** INSERTING "|" BETWEEN "READ" AND "BRAVO" ON LINE 7
   8 |  1|  56 |ALPHA := BRAVO;   CHARLIE : 2 * ALPHA;

** CORRECTION ** DELETING ";" ON LINE 8
   9 |  1|  61 |WRITE ("ALPHA=",A)

** CORRECTION ** REPLACING ":" BY "<SETQ>" ON LINE 8

*** ERROR, UNDECLARED IDENTIFIER.  LAST PREVIOUS ERROR WAS DETECTED ON LINE 0 **
          A DECLARED TYPE SIMPLE, INITIAL VALUE = GARBAGE
  10 |  1|  72 |WRITE ("BRAVO=", BRAVO);
  11 |  1|  73 |FOR ALPHA := 1 STEP 1 UNTIL 2 ZZ WRITE ("A");
```

Figure 6

```
** CORRECTION ** INSERTING ";" BETWEEN "READ" AND "WRITE" ON LINE 10
  12 | 1|  95 |DELTA <5 := 4;

** CORRECTION ** REPLACING "ZZ" BY "THEN" ON LINE 11

** CORRECTION ** REPLACING "THEN" BY "DO" ON LINE 11
  13 | 1| 109 |CHARLIE 5 :=;

** CORRECTION ** INSERTING ">" BETWEEN "5" AND "<SETQ>" ON LINE 12
  14 | 1| 114 |IF BRAVO GARBAGE CHARLIE THEN MOREGARBAGE;

** CORRECTION ** TRANSPOSING "5" AND "<SETQ>" ON LINE 13
  15 | 1| 117 |END;

*** SECOND ERROR LOCATED WHILE REVERSE PARSING.
    ILLEGAL SYMBOL PAIR: MOREGARBAGE THEN

THE FORWARD PARSING ERROR IS:

*** ERROR, ILLEGAL SYMBOL PAIR: <IDENTIFIER> <IDENTIFIER>.  LAST PREVIOUS ERROR
          WAS DETECTED ON LINE 9 **
PARTIAL PARSE SO FAR IS:  "<BLOCK HEAD> <BLOCK BODY> ; IF <IDENTIFIER> "

** SORRY ABOUT THAT, BUT I CAN'T FIX IT...GOOD LUCK.

** RESUME **
  16 | 0| 124 |EOF

** CORRECTION ** DELETING ";" ON LINE 16
PRT=11, DATA=12, CODE=31 (WORDS).
CODE FILE WRITTEN
END OF COMPILATION JUNE 2, 1972.  CLOCK TIME = 3:38:53.68.

16 CARDS WERE READ.
2 ERRORS (0 SEVERE) WERE DETECTED.
THE LAST DETECTED ERROR WAS ON LINE 15.
```

Figure 6 (Cont'd)

that the corrections made were those that a human reader would be expected to make. A few configurations were made syntactically correct but not in the logical sense defined above.

Example: FOR A := PETS 1 1 UNTIL...

PETS, not recognized as a misspelling of STEP, was interpretted as an identifier resulting in the first "1" being replaced by STEP.

For the case of self-embedding symbol pairs such as BEGIN...END and (...), the omission or duplication of the leading or left symbol resulted in the deletion or insertion of the right symbol at a later point in the input stream. At first brush, this particular correction may seem fairly gross but the delection/insertion points were syntactically defined without regard for what ever the programmer's intended logic may have been.

For those errors that the system could not correct, the history of the attempts at solution prior to abandoning the error and deleting code and a definition of the last error encountered by the reverse parser were made available to the programmer, thereby fairly isolating the error and defining the inability to make a correction.

The time involved in correcting errors averaged about 0.015 seconds per error for the programs tested.

V. CONCLUSIONS

The syntax error correcting procedure proposed in this thesis is a viable system. While costs in terms of time and space are involved, its effects on a user's code are considerably more attractive than those of popular recovery systems employing automatic deletion of code to some stop symbol. Whereas the proposed system was defined to be grammar independent, the working model implemented was semi-automatic, using predefined start states for the reverse and forward parsers. It is recognized that these crossover points are significant with respect to fully automating the error correction process; however, they are the only points in the model that are language dependent. The correction procedures themselves are language independent; their only parameters are parse states and associated symbol sets defined by the parser constructor.

The power in the procedure is attibutable to the LR(k) parsing employed. Errors are examined in a very large context provided by the two disjoint state stacks of the forward and reverse parsers. Through LR(k) parsing, syntax errors are detected as the input stream is read and are precluded from the symbol stack.

The model demonstrates that the proposed system detects and deterministically corrects a large class of errors thereby affording the programmer maximum exposure of his code to the analytical processes. A strong heuristic attempt to correct is provided for those cases that the error cannot be resolved deterministically. Should error correction fail entirely, the system provides a good diagnosis and all residue of

the error is removed, thereby insuring against generating or cascading syntax errors through the remainder of the input stream.

VI. EXTENSIONS

The error correction system described in this thesis indicates several areas where worthwhile extensions can be made and where further analysis is necessary.

A. KEY DEFINITION

As keys seem to lend themselves to empirical definition then it would seem logical that they may be analytically defined as the grammar to which they belong is being analyzed. An analyzer capable of defining a set of valid keys should also enable automating the error corrector by associating keys with states for both parsers and providing an automatic link to a key and the engagement of the error analyzing system from any state in the parser when a syntax error is detected. It may be feasible and practical to define a hierarchy of keys so that it would not be required to go beyond a minimum distance past the outer limit of the allowable error sequence. This would serve to minimize the likelihood of encountering another error thereby causing the corrector to abort.

It may also be of value to define a grammar analyzer capable of recognizing hierarchies of key symbols and symbol strings and associating these sets with unique parser states such that, for a given sentential form, dedicated keys are available to minimize the key-to-error distance and increase the probability that a key itself does not constitute an error.

B. ERROR EXTENSION

The current implementation severely restricts errors to single symbols except in the case of adjacent symbol transposition. A logical extension would be to extend the limits to provide for multiple symbol errors. This would require either predefining and storing the legal symbol strings or defining a symbol string generator to be called as required.

C. CLASSIC LR(k) VERSUS SLR(k)

The classic LR(k) parser stops whenever it encounters an error symbol in either a read or lookahead state. The parser employed in the model defaults to the next read state in the event that the lookahead symbol is not a member of the symbol set associated with a particular lookahead state. That is, a successful lookahead defines a stack reduction, otherwise the decision is to stack (read) the lookahead symbol via the next logical read state. Only after the symbol is read is it determined that it is an error symbol or not. It would be advantageous to be able to stop the parser in a lookahead state rather than in the next read state so as to keep the symbol preceding the error readily accessible at the top of the stack and available to participate in error analysis.

D. STACK ACCESSIBILITY

As inconvenient as it may be, there are constructs in the grammar such that their containing errors is undetectable until the point where correction is needed is in the stack. More analysis is needed to weigh the costs of incorporating a means of accessing the stack and, if necessary, deleting and regenerating code against the desire to and benefits of being able to correct this type error.

ALGOL-E (MODIFIED)

```
<PROGRAM> ::= <BLOCK>
<BLOCK> ::= <BLOCK HEAD> <BLOCK END>
<BLOCK HEAD> ::= <BEGIN> <DECLARATION SET> .
               |  <BEGIN>
<BEGIN> ::= BEGIN
<DECLARATION SET> ::= <DECLARATION>
                    |  <DECLARATION SET> . <DECLARATION>
<BLOCK END> ::= <BLOCK BODY> END
              |  <BLOCK BODY> ; END
<BLOCK BODY> ::= <STATEMENT>
               |  <BLOCK BODY> ; <STATEMENT>
<STATEMENT> ::= <IF STATEMENT>
              |  <SIMPLE STATEMENT>
<SIMPLE STATEMENT> ::= <ASSIGNMENT STATEMENT>
                     |  <BLOCK>
                     |  <FOR STATEMENT>
                     |  <WHILE STATEMENT>
                     |  <READ STATEMENT>
                     |  <WRITE STATEMENT>
                     |  <CASE STATEMENT>
                     |  <PROCEDURE CALL>
                     |  <SIMPLE PROCEDURE CALL>
```

```
<DECLARATION> ::= <SIMPLE DECLARATION>
                | <ARRAY DECLARATION>
                | <SUBPROGRAM DECLARATION>
                | <FORWARD DECLARATION>
<SIMPLE DECLARATION> ::= <LOCAL HEAD> <IDENTIFIER>
<LOCAL HEAD> ::= LOCAL
               | <LOCAL HEAD> <IDENTIFIER> ,
<ASSIGNMENT STATEMENT> ::= <VARIABLE> <RIGHT PART>
<RIGHT PART> ::= <SETQ> <EXPRESSION>
               | <SETQ> <VARIABLE> <RIGHT PART>
<VARIABLE> ::= <IDENTIFIER>
             | <SUBSCRIPTED VARIABLE>
<SUBSCRIPTED VARIABLE> ::= <SUBSCRIPT HEAD> <EXPRESSION> >
<SUBSCRIPT HEAD> ::= <IDENTIFIER> <
                   | <SUBSCRIPT HEAD> <EXPRESSION> ,
<EXPRESSION> ::= <ARITHMETIC EXPRESSION>
               | <IF EXPRESSION> <EXPRESSION>
<IF EXPRESSION> ::= <IF CLAUSE> <EXPRESSION> ELSE
<ARITHMETIC EXPRESSION> ::= <TERM>
                          | <ARITHMETIC EXPRESSION> + <TERM>
                          | <ARITHMETIC EXPRESSION> - <TERM>
                          | - <TERM>
                          | + <TERM>
<TERM> ::= <PRIMARY>
```

```
          |   <TERM> * <PRIMARY>
          |   <TERM> / <PRIMARY>
<PRIMARY> ::= <PRIMARY ELEMENT>
            |   <PRIMARY> ' <PRIMARY ELEMENT>
<PRIMARY ELEMENT> ::= <VARIABLE>
                    |   <NUMBER>
                    |   <FUNCTION CALL>
                    |   ( <ASSIGNMENT STATEMENT> )
                    |   ( <EXPRESSION> )
<SIMPLE PROCEDURE CALL> ::= CALL <IDENTIFIER>
<PROCEDURE CALL> ::= CALL <CALL HEADING> <EXPRESSION> )
<FUNCTION CALL> ::= <CALL HEADING> <EXPRESSION> )
<CALL HEADING> ::= <IDENTIFIER> (
                 |   <CALL HEADING> <EXPRESSION> ,
<BOOLEAN EXPRESSION> ::= <BOOLEAN TERM>
                       |   <BOOLEAN EXPRESSION> OR <BOOLEAN TERM>
<BOOLEAN TERM> ::= <BOOLEAN PRIMARY>
                 |   NOT <BOOLEAN PRIMARY>
                 |   <BOOLEAN TERM> AND <BOOLEAN PRIMARY>
<BOOLEAN PRIMARY> ::= <LOGICAL EXPRESSION>
                    |   ( <BOOLEAN EXPRESSION> )
<LOGICAL EXPRESSION> ::= <EXPRESSION> <RELATION> <EXPRESSION>
<RELATION> ::= LSS
             |   LEQ
```

```
                |   EQL
                |   NEQ
                |   GEQ
                |   GTR
<ARRAY DECLARATION> ::= <ARRAY LIST> <BOUND PAIR LIST>
<ARRAY LIST> ::= <ARRAY HEAD> <IDENTIFIER>
<ARRAY HEAD> ::= ARRAY
                  |   <ARRAY HEAD> <IDENTIFIER> ,
<BOUND PAIR LIST> ::= <BOUND PAIR HEAD> <BOUND PAIR> >
<BOUND PAIR HEAD> ::= <
                      |   <BOUND PAIR HEAD> <BOUND PAIR> ,
<BOUND PAIR> ::= <LOWER BOUND> : <EXPRESSION>
<LOWER BOUND> ::= <EXPRESSION>
<SUBPROGRAM DECLARATION> ::= <SUBPROGRAM HEADING> . <STATEMENT>
<SUBPROGRAM HEADING> ::= <FUNCTION HEADING>
                         |   <PROCEDURE HEADING>
<FUNCTION HEADING> ::= <PARAMLESS FUNCTION>
                       |   <FUNCTION&PARAMS>
<PROCEDURE HEADING> ::= <PARAMLESS PROC>
                        |   <PROC&PARAMS>
<PARAMLESS FUNCTION> ::= FUNCTION <IDENTIFIER>
<FUNCTION&PARAMS> ::= <FUNCTION HEAD> <IDENTIFIER> )
<FUNCTION HEAD> ::= FUNCTION <IDENTIFIER> (
                    |   <FUNCTION HEAD> <IDENTIFIER> ,
```

```
<PARAMLESS PROC> ::= PROCEDURE <IDENTIFIER>
<PROC&PARAMS> ::= <PROCEDURE HEAD> <IDENTIFIER> )
<PROCEDURE HEAD> ::= PROCEDURE <IDENTIFIER> (
                   |   <PROCEDURE HEAD> <IDENTIFIER> ,
<FORWARD DECLARATION> ::= <FORWARD FUNCTION>
                        |   <FORWARD PROCEDURE>
<FORWARD FUNCTION> ::= <FORWARD FUNC HEAD> <IDENTIFIER>
<FORWARD FUNC HEAD> ::= FORWARD FUNCTION
                      |   <FORWARD FUNC HEAD> <IDENTIFIER> ,
<FORWARD PROCEDURE> ::= <FORWARD PROC HEAD> <IDENTIFIER>
<FORWARD PROC HEAD> ::= FORWARD PROCEDURE
                      |   <FORWARD PROC HEAD> <IDENTIFIER> ,
<WHILE STATEMENT> ::= <WHILE CLAUSE> <DO STATEMENT>
<WHILE CLAUSE> ::= <WHILE> <BOOLEAN EXPRESSION>
<WHILE> ::= WHILE
<CASE STATEMENT> ::= <CASE HEADING> <BLOCK>
<CASE HEADING> ::= CASE <EXPRESSION> OF
<IF STATEMENT> ::= <IF GROUP>
                 |   <IF ELSE GROUP> <STATEMENT>
<IF GROUP> ::= <IF CLAUSE> <STATEMENT>
<IF ELSE GROUP> ::= <IF CLAUSE> <STATEMENT> ELSE
<IF CLAUSE> ::= IF <BOOLEAN EXPRESSION> THEN
<FOR STATEMENT> ::= <FOR CLAUSE> <STEP EXPRESSION> <UNTIL CLAUSE> <DO STATEMENT>
<FOR CLAUSE> ::= FOR <ASSIGNMENT STATEMENT>
```

```
<STEP EXPRESSION> ::= <STEP> <EXPRESSION>
<STEP> ::= STEP
<UNTIL CLAUSE> ::= UNTIL <EXPRESSION>
<DO STATEMENT> ::= DO <SIMPLE STATEMENT>
<READ STATEMENT> ::= <READ HEAD> <VARIABLE> |
<READ HEAD> ::= READ |
                |   <READ HEAD> <VARIABLE> ,
<WRITE STATEMENT> ::= <WRITE HEAD> <EXPRESSION> )
                    |   <WRITE HEAD> <STRING> )
                    |   <WRITE HEAD> <TAB EXPRESSION> )
<WRITE HEAD> ::= WRITE (
                |   WRITEON (
                |   <WRITE HEAD> <EXPRESSION> ,
                |   <WRITE HEAD> <STRING> ,
                |   <WRITE HEAD> <TAB EXPRESSION> ,
<TAB EXPRESSION> ::= TAB <EXPRESSION>
```

```
SYNTACTICAL_ANALYZER:
   PROCEDURE;

   DECLARE /* SP POINTS TO THE ELEMENT ABOVE THE TOP OF THE STACK.  */
      STATE_STACK(255) BIT(16) INITIAL(0),
      STATE_#  BIT(8) INITIAL (0),
      /* THE START STATE IS READ_LINEAR 0  */
      (I, J)  BIT(16),
      (P_SYM, N_SYM, MSG) CHARACTER;

   DECLARE /* VARIABLES USED FOR SPELLING CORRECTIONS  */
      STATE_BEFORE_READ BIT(16),
      DUMMY_TOKEN FIXED,
      (TSP, SAVE_SP) BIT(8),
      RESET_1 BIT(1) INITIAL (0),  /*  RESTART IN PREVIOUS STATE  */
      RESET_2 BIT(1) INITIAL (0);   /*  RESTART TWO STATES BACK  */

   DECLARE  /*  VARIABLES USED FOR R_PARSING SPELLING & ERROR CORRECTION  */
      I_SYM BIT(8),  /*  SYMBOL TO BE INSERTED WHEN DEFINED VIA REV PARSING  */
      (I_B4_ST, I_B4_TOKEN) BIT(1),
      R_STATE_STACK(255) BIT(16),
      (RP, SAVE_RP, R_SP, SP1, SP2) FIXED,
      R_STATE_# BIT(8),
      R_STATE_BEFORE_READ BIT(16),
      (R_TOKEN, R_DUMMY_TOKEN) FIXED,
      RP_ERROR BIT(1) INITIAL(0),
      (SFS_#,SRS_#) BIT(8),  /*  FWD & REV PARSER STATE_# BEFORE ERROR  */
      (S_TYPE,R_S_TYPE) BIT(8),  /* FWD & REV PARSER STATE TYPE BEFORE ERROR  */
      R_P_S BIT(1) INITIAL(0),
      TEMP_TOP BIT(8) INITIAL (1),  /*  1 = "EOF"  */
      HAVE_ALREADY_MADE_INSERTION BIT(1) INITIAL(0),  /*  FALSE  */
      #_FIXITS FIXED INITIAL(0),  /* 4 IS MAX FIXUP TRIES THEN DELETE CODE  */
      DELETING BIT(1) INITIAL(0),  /* FLAG, INIT FALSE  */

      /*  SOME LITERALS...*/
      INSERT_I_BEFORE_TOKEN LITERALLY  'I_SYM = I;  I_B4_TOKEN = TRUE',
      INSERT_I_BEFORE_STACK_TOP LITERALLY  'I_SYM = I;  I_B4_ST = TRUE',
      RESTART LITERALLY 'RETURN TRUE',
      R_NEXT_STATE LITERALLY
      'SRS_#,R_STATE_BEFORE_READ = R_STATE_STACK(RP);
      R_STATE_STACK(RP),R_STATE_#',
      R_STATE_TYPE LITERALLY 'SHR(R_STATE_STACK(RP),8)',
      DELETE_SP LITERALLY
```

```
    'OUTPUT = ''''; OUTPUT = ''** CORRECTION ** DELETING "'' || CK_SYM(SP) ||
     '''" ON LINE '' ||SAVE_CARD_#;  NEXT_SYMBOL(SP) = 255;  DELETING = TRUE',
    DELETE_SP_PLUS_1 LITERALLY
    'OUTPUT=''''; OUTPUT=''** CORRECTION ** DELETING "''||CK_SYM(SP+1)||
    '''" ON LINE '' || SAVE_CARD_#;
      NEXT_SYMBOL(SP + 1) = 255;  DELETING = TRUE';

DECLARE NEXT_STATE LITERALLY
    'SES_#,STATE_BEFORE_READ = STATE_STACK(SP);  STATE_STACK(SP), STATE_#',
    STATE_TYPE  LITERALLY 'SHR(STATE_STACK(SP), 8)',
     NEXT_SYMBOL(256) BIT(8);

DECLARE STATE BIT(16);  /* USED IN PROCEDURE TRACER, BELOW  */

PUSH_AND_READ:
   PROCEDURE;
      IF SP < 255
      THEN SP = SP + 1;
      ELSE
      DO;
         CALL ERROR('PARSE STACK OVERFLOW.  COMPILATION TERMINATED.');
         GO TO END_OF_ALL_OF_IT;
      END;
      IF ¬REVERSE_PARSING THEN
         DO;  NEXT_SYMBOL(256) = TEMP_TOP;  /*  SAVE TOP-1  */
         TEMP_TOP, NEXT_SYMBOL(SP) = TOKEN;
         END;
      ELSE NEXT_SYMBOL(SP) = TOKEN;
      VAR(SP) = BCD;  FIXV(SP) = NUMBER_VALUE;
      IF SCANNER_TURNED_OFF THEN RETURN;
      CALL SCAN;
   END PUSH_AND_READ;


TRACER:  PROCEDURE(BOTTOM, TOP, MSG);
   DECLARE (BOTTOM, TOP) FIXED, MSG CHARACTER;
   DO I = BOTTOM TO TOP;    /* OF PARSE STACK  */
         STATE = STATE_STACK(I);
         IF STATE <= "1FF"     /* IS STATE A READ STATE?  */
         THEN P_SYM = V(SYM_BEFORE_READ("FF" & STATE));
         ELSE P_SYM = V(SYM_BEFORE_LA("FF" & STATE));
         IF LENGTH(MSG) > 105 THEN
         DO;
            OUTPUT = MSG;   MSG = '    ';
         END;
         MSG = MSG || ' ' || P_SYM;
      END;
      MSG = MSG || ' "';
```

```
      OUTPUT = MSG;
   END TRACER;

STEPPING:  PROCEDURE;
/*  CALLED FROM REVERSE_PARSING_SUCCESSFUL.  STEP THE R_PARSER 'TIL
    NEXT READ OR LA STATE   */
   AGAIN:  DO CASE(R_STATE_TYPE);
      /*0*/;  /*1*/;
      DO;  RP = RP - R_#_TO_POP(R_STATE_#);
      R_NEXT_STATE = R_REDUCE_SUCC(R_STATE_#);
      END;
      /*3*/;  /*4*/;
      DO;  I = R_LB_START(R_STATE_#);
      J = I + R_LB_#(R_STATE_#);
      DO WHILE R_STATE_STACK(RP - 1) ¬= R_LB_STATE(I) & I < J;
         I = I + 1;   END;
      R_NEXT_STATE = R_RESUME_STATE(I);
      END;
   END;  /*  CASE  */
   IF R_STATE_TYPE ¬= 0 & R_STATE_TYPE ¬= 3 THEN GO TO AGAIN;
   SRS_# = R_STATE_STACK(RP);
END;  /* STEPPING  */

FP_INTRSCT_RP:  PROCEDURE(SFS_#, SRS_#, SET) BIT(8);
/*  DETERMINE THE INTERSECTION OF LEGAL SYMBOLS FOR
    THE FORWARD AND REVERSE PARSERS.  IF THE INTERSECTION
    IS EMPTY RETURN 0 ELSE RETURN V SUBSCRIPT OF TERMINAL
    SYMBOL IN THE INTERSECTION
      SET 0:  FORWARD & REVERSE PARSERS IN READ STATES
      SET 1:  FP IN LA STATE, RP IN READ STATE
      SET 2:  FP IN READ STATE, RP IN LA STATE
      SET 3:  BOTH PARSERS IN LA STATES                    */
   DECLARE (SFS_#,SRS_#,SET,BYTE_#,SHR_#) BIT(8),
           (C,D,W,X,Y,Z) BIT(16),
           (LA_SET,R_LA_SET) BIT(50);
   LA_SET = LA_TABLE(SFS_#);  R_LA_SET = R_LA_TABLE(SRS_#);
   W = READ_START(SFS_#);  X = W + RD_#(SFS_#) - 1;
   Y = R_READ_START(SRS_#);  Z = Y + R_RD_#(SRS_#) - 1;
   DO CASE(SET);

      DO;
      DO C = W TO X;
         DO D = Y TO Z;
            IF SYM_LIST(C) = R_SYM_LIST(D) THEN RETURN SYM_LIST(C);
         END;
      END;
```

```
            RETURN 0;
            END;  /* CASE 0  */

            DO;
            DO D = Y TO Z;
               BYTE_# = SHR(R_SYM_LIST(D),3);
               SHR_# = 7 - (R_SYM_LIST(D) & "(1)111");
               IF SHR(BYTE(LA_SET,BYTE_#),SHR_#) THEN
                  RETURN R_SYM_LIST(D);
            END;
            RETURN 0;
            END;  /*  CASE 1  */

            DO;
            DO C = W TO X;
               BYTE_# = SHR(SYM_LIST(C),3);
               SHR_# = 7 - (SYM_LIST(C) & "(1)111");
               IF SHR(BYTE(R_LA_SET,BYTE_#),SHR_#) THEN
                  RETURN SYM_LIST(C);
            END;
            RETURN 0;
            END;  /*  CASE 2  */

            DO;
            DO D = 0 TO #_TERMINALS;
               C = #_TERMINALS - D;
               BYTE_# = SHR(C,3);
               SHR_# = 7 - (C & "(1)111");
               IF SHR(BYTE(LA_SET,BYTE_#),SHR_#)
                  & SHR(BYTE(R_LA_SET,BYTE_#),SHR_#)
                     THEN RETURN C;
            END;
            RETURN 0;
            END;  /*  CASE 3  */
         END;  /* CASE STATEMENT  */
      END;  /*  FS_INTRSCT_RS  */

      CK_SYM:  PROCEDURE(PTR) CHARACTER;
         /*  RETURNS LITERAL VALUE OF STRINGS, IDENTS, AND NUMBERS
             TO CORRECTION DIAGNOSTICS  */
         DECLARE PTR FIXED, N_S BIT(8);
         N_S = NEXT_SYMBOL(PTR);
         IF N_S = STRING | N_S = IDENT THEN RETURN VAR(PTR);
         IF N_S = NUMBER THEN RETURN FIXV(PTR); ELSE RETURN V(N_S);
      END;  /*   CK_SYM  */
```

```
TRANSPOSE:  PROCEDURE(PTR_1,PTR_2);
   /*  TRANSPOSES STACK SYMBOLS AT PTR_1 AND PTR_2  */
   DECLARE (PTR_1, PTR_2, F_V) FIXED, V CHARACTER, N_S BIT(8);
   OUTPUT = '';
   OUTPUT = '** CORRECTION ** TRANSPOSING "' || CK_SYM(PTR_1) || '" AND "'
            || CK_SYM(PTR_2) || '" ON LINE ' || SAVE_CARD_#;
   V=VAR(PTR_1);  F_V = FIXV(PTR_1);
   VAR(PTR_1) = VAR(PTR_2);  FIXV(PTR_1) = FIXV(PTR_2);
   VAR(PTR_2) = V;  FIXV(PTR_2) = F_V;
   N_S = NEXT_SYMBOL(PTR_1);  NEXT_SYMBOL(PTR_1) = NEXT_SYMBOL(PTR_2);
   NEXT_SYMBOL(PTR_2) = N_S;
END;  /*  TRANSPOSE  */

REPLACE:  PROCEDURE (PTR, I_SYM);
   /*  SYMBOL(PTR) IS REPLACED BY INTERSECTION SYMBOL  */
   DECLARE PTR FIXED,  I_SYM BIT(16);
   OUTPUT = '';
   OUTPUT = '** CORRECTION ** REPLACING "' || CK_SYM(PTR) || '" BY "'
            || V(I_SYM) || '" ON LINE ' || SAVE_CARD_#;
   NEXT_SYMBOL(PTR) = I_SYM;
   VAR(PTR) = V(I_SYM);
END;  /* REPLACE  */

SET_MEMBER:  PROCEDURE(PTR, SET) BIT(1);
   /*  IF SYM IS A MEMBER OF SET THEN RETURN TRUE ELSE FALSE
         SET 0:  FORWARD PARSER READ SET
         SET 1:     "        "    LA SET
         SET 2:  REVERSE     "    READ SET
         SET 3:     "        "    LA SET                    */
   DECLARE  PTR FIXED, (SYM,SET) BIT(8),
           (A,B,C,D) BIT(16), LA_SET BIT(50),
           (BYTE_#,SHR_#) BIT(8);
   SYM = NEXT_SYMBOL(PTR);
   DO CASE(SET);

      DO;  A = READ_START(SFS_#);  B=A + RD_#(SFS_#) - 1;
      DO C = A TO B;
         IF SYM = SYM_LIST(C) THEN RETURN TRUE;
      END;  RETURN FALSE;
      END;  /*  CASE 0  */

      DO;  BYTE_# = SHR(SYM,3);  SHR_# = 7 - (SYM & "(1)111");
      IF SHR(BYTE(LA_TABLE(SFS_#),BYTE_#),SHR_#) THEN
         RETURN TRUE; ELSE RETURN FALSE;
      END;  /*  CASE 1  */
```

```
      DO;  A = R_READ_START(SRS_#);  B = A + R_RD_#(SRS_#) - 1;
      DO C = A TO B;
         IF SYM = R_SYM_LIST(C) THEN RETURN TRUE;
      END;  RETURN FALSE;
      END;  /*  CASE 2  */

      DO;  BYTE_# = SHR(SYM,3);  SHR_# = 7 - (SYM & "(1)111");
      IF SHR(BYTE(R_LA_TABLE(SRS_#),BYTE_#),SHR_#) THEN
         RETURN TRUE; ELSE RETURN FALSE;
      END;  /* CASE 3  */
   END;  /*  CASE STATEMENT  */
END;  /*  SET_MEMBER  */


CHECK_CONTEXT_OF_TOP_AND_TOKEN:  PROCEDURE(A,B,C,D) BIT(1);
   /*  CHECKS VARIOUS CONFIGURATIONS OF TOP AND TOKEN WITH RESPECT
       TO FP AND RP  */
   DECLARE (A,B,C,D) BIT(8);
   I = SET_MEMBER(SP + 1,A);  J = SET_MEMBER(SP,B);
   IF I = TRUE & J = TRUE THEN
      DO;  CALL TRANSPOSE(SP,SP + 1);  RESTART;  END;
   I = FP_INTRSCT_RP(SFS_#,SRS_#,C);
   IF I > 0 THEN
      DO;  CALL REPLACE(SP,I);  DELETE_SP_PLUS_1;  RESTART;  END;
   ELSE IF SET_MEMBER(SP + 1,D) THEN
      DO;  DELETE_SP;  RESTART;  END;
   ELSE RETURN FALSE;
END; /*  CHECK_CONTEXT_OF_TOP_AND_TOKEN  */


CAN_DO_WITHOUT_TOP:  PROCEDURE(F_TYPE) BIT(1);
   DECLARE (F_TYPE,X) BIT(8);
   IF R_S_TYPE = 0 THEN X = 2; ELSE X = 3;  /*  RP IN READ OR LA  */
   IF SET_MEMBER(SP+1,F_TYPE) THEN
      DO;  I = SET_MEMBER(SP-1,X);  J = SET_MEMBER(256,X);
      IF I = TRUE & J = TRUE THEN
         DO;  DELETE_SP;  RETURN TRUE;  END;
      END;
   RETURN FALSE;
END;  /* CAN_DO_WITHOUT_TOP  */


MEMBER_FP_LA:  PROCEDURE BIT(8);
   /*  RETURNS A MEMBER OF CURRENT FP LOOKAHEAD SYMBOL SET  */
   DECLARE (A,BYTE_#,SHR_#) BIT(8);
   DO A = 0 TO #_TERMINALS;
   BYTE_# = SHR(A,3);
   SHR_# = 7 - (A & "7");
```

```
   IF SHR(BYTE(LA_TABLE(SFS_#),BYTE_#),SHR_#) THEN RETURN A;
   END;
END;  /* MEMBER_FP_LA  */


                    /************************
                     *                      *
                     *    REVERSE PARSER    *
                     *                      *
                     ************************/


R_PARSING:  PROCEDURE;
   REVERSE_PARSING = TRUE;  /*  REVERSE_PARSING: A GLOBAL VARIABLE  */
   /*  INITIALIZE REVERSE_PARSER...  */
   RP = 0;
   IF R_P_S THEN
      DO;  R_SP = SAVE_SP + 1;  #_FIXITS = #_FIXITS + 1;  GO TO INIT;  END;
   ELSE DO;  R_SP = SP + 1;
      IF NEXT_SYMBOL(SP) = EOFILE THEN GO TO SWITCH;
      CALL PUSH_AND_READ;
      R_SP = R_SP + 1;
      DO WHILE NEXT_SYMBOL(SP) ¬= SEMI & NEXT_SYMBOL(SP) ¬= EOFILE &
         NEXT_SYMBOL(SP) ¬= DOT;
         CALL PUSH_AND_READ;
         R_SP = R_SP + 1;
         END;
      END;
   /*  SAVE_SP MARKED LIMIT OF FORWARD PARSER, SWITCH SP AND SAVE_SP  */
   SWITCH:  I = SP;  SP = SAVE_SP;  SAVE_SP = I;
   INIT:  IF NEXT_SYMBOL(SAVE_SP) = EOFILE THEN
      R_STATE_STACK(0),R_STATE_#,SRS_#,R_STATE_BEFORE_READ = 0;
   ELSE IF NEXT_SYMBOL(SAVE_SP) = SEMI THEN
      R_STATE_STACK(0),R_STATE_#,SRS_#,R_STATE_BEFORE_READ = 4;
   ELSE R_STATE_STACK(0),R_STATE_#,SRS_#,R_STATE_BEFORE_READ = 3;
   DO WHILE R_SP >  SP;  /*  ...OR UNTIL RETURN WITH ERROR  */
   DO CASE(R_STATE_TYPE);

      DO;  /*  READ VIA LINEAR SEARCH  */
      IF RESET_2 THEN
         DO;  RP = RP + 1;  R_SP = R_SP - 1;
         IF RP = SAVE_PP THEN
            DO;  RESET_2 = FALSE;  NEXT_SYMBOL(R_SP) = R_DUMMY_TOKEN;  END;
         END;
      ELSE IF RESET_1 THEN
         DO;  RP = RP + 1;  RESET_1 = FALSE;
         NEXT_SYMBOL(R_SP) = R_DUMMY_TOKEN;  END;
```

```
ELSE
   DO;  RP = RP + 1;  R_SP = R_SP - 1;
   R_TOKEN = NEXT_SYMBOL(R_SP - 1);  END;
I = R_READ_START(R_STATE_#);
J = I + R_RD_#(R_STATE_#);
DO WHILE NEXT_SYMBOL(R_SP) ¬= R_SYM_LIST(I) & I < J;
   I = I + 1;  END;
/*  CHECK FOR MISSPELLED "END"...  */
IF R_TOKEN = IDENT THEN
   DO;  K = CHECK_RES_WORD_SPELLING(VAR(R_SP-1));
   IF K > 0 THEN
      DO;  R_TOKEN,NEXT_SYMBOL(R_SP-1) = K; VAR(R_SP-1) = V(K);  END;
   END;
IF SHR(R_STATE_LIST(I),8) = 6 THEN RP_ERROR = TRUE;
IF SHR(R_STATE_LIST(I),8) = 6 & SHR(R_STATE_BEFORE_READ,8) = 3 THEN
   DO;  R_STATE_STACK(RP), R_STATE_# = R_STATE_LIST(I);
   END;
ELSE DO;  SRS_#, R_STATE_BEFORE_READ = R_STATE_STACK(RP - 1);
   R_STATE_STACK(RP),  R_STATE_# = R_STATE_LIST(I);
   END;
END;

DO;  /*  READ VIA AN ARRAY CASE: NOT USED  */  END;

DO;  /*  REDUCE  */
RP = RP - R_#_TO_POP(R_STATE_#);
R_NEXT_STATE = R_REDUCE_SUCC(R_STATE_#);
END;

DO;  /*  LOOK AHEAD ORDINARY  */
DECLARE(R_LA_SYMBOL,BYTE_#,SHR_#) BIT(8);
IF RESET_1 | RESET_2 THEN R_LA_SYMBOL = R_DUMMY_TOKEN;
   ELSE R_LA_SYMBOL = R_TOKEN;
/*  NEED A COUPLE OF FIXITS HERE  */
IF (R_STATE_#=8 & R_TOKEN=COMMA) THEN
   DO;  R_NEXT_STATE = R_FAIL_STATE(R_STATE_#);  GO TO HANDLED;  END;
BYTE_# = SHR(R_LA_SYMBOL,3);
SHR_# = 7 - (R_LA_SYMBOL & "(1)111");
IF SHR(BYTE(R_LA_TABLE(R_STATE_#),BYTE_#),SHR_#)
THEN DO;  R_NEXT_STATE = R_SUCC_STATE(R_STATE_#);  END;
ELSE DO;  R_NEXT_STATE = R_FAIL_STATE(R_STATE_#);  END;
HANDLED:  ;
END;

DO;  /*  LOOK AHEAD (FOR A PROD WITH EMPTY RT PART)  */  END;

DO;  /*  LOOK BACK  */
DECLARE R_PREV_STATE BIT(16);
```

```
      R_PREV_STATE = R_STATE_STACK(RP - 1);
      I = R_LB_START(R_STATE_#);
      J = I + R_LB_#(R_STATE_#);
      DO WHILE R_PREV_STATE ¬= R_LB_STATE(I) & I < J;
         I = I + 1;  END;
      R_NEXT_STATE = R_RESUME_STATE(I);
      END;

      DO;  /*  ERROR  */
      SAVE_RP = RP;
      IF NEXT_SYMBOL(R_SP + 1) = IDENT THEN
         DO;  I = CHECK_RES_WORD_SPELLING(VAR(R_SP+1));
         IF I > 0 THEN  /* FOUND IT  */
            DO;  RESET_2 = TRUE;  TSP = R_SP + 1;
            NEXT_SYMBOL(TSP) = I;  VAR(TSP) = V(I);
            FIXV(TSP) = 0;  R_DUMMY_TOKEN = NEXT_SYMBOL(R_SP);
            RP = RP - 2;  R_SP = R_SP + 2;
            R_STATE_# = R_STATE_STACK(RP);
            GO TO R_DONE;
            END;
         END;
      IF NEXT_SYMBOL(R_SP) = IDENT THEN
         DO;  I = CHECK_RES_WORD_SPELLING(VAR(R_SP));
         IF I > 0 THEN
            DO;
            NEXT_SYMBOL(R_SP) = I;
            VAR(R_SP) = V(I);  FIXV(R_SP) = 0;  R_DUMMY_TOKEN = I;
            RP = RP - 1;  RESET_1 = TRUE;
            IF SHR(R_STATE_BEFORE_READ,8) = 3 THEN  /* LA STATE  */
               R_STATE_STACK(RP), R_STATE_# = R_STATE_BEFORE_READ;
            ELSE
               R_STATE_# = R_STATE_STACK(RP);
            GO TO R_DONE;
            END;  /*  IF I > 0...  */
         END;  /*  IF NEXT_SYMBOL(R_SP) = ...  */
      RETURN;
      R_DONE:  RP_ERROR = FALSE;
      END;  /*  ERROR CASE  */
   END;  /*  CASE(R_STATE_TYPE)  */
   END;  /*  WHILE R_SP = ...  */
END;  /*  R_PARSING  */


ERROR_ANALYZER:  PROCEDURE BIT(1);
   DECLARE (W,X,Y,Z) BIT(8);
   W = 0;  X = 1;  Y = 2;  Z = 3;
   CALL R_PARSING;
   REVERSE_PARSING = FALSE;
```

```
    S_TYPE = SHR(STATE_BEFORE_READ,8);  R_S_TYPE = SHR(R_STATE_BEFORE_READ,8);
    IF R_SP > SP THEN GO TO H3;  /*  PTRS DON'T MATCH  */
    IF RP_ERROR THEN GO TO H2;
    /*  POSSIBLE CONDITIONS AT THIS POINT:
        FP: READ, E          RP: READ OR LA, ¬E
            LA, E                READ OR LA, ¬E                */
    IF S_TYPE = 3 THEN GO TO H1;  /*  LA STATE  */
    /*  REVERSE PARSER STOPPED WITH PTRS MATCHED AT STACK TOP...  */
    IF CAN_DO_WITHOUT_TOP(W) = TRUE THEN RESTART;
    IF R_S_TYPE = 0 THEN
       I1 = FP_INTRSCT_RP(SFS_#,SRS_#,W);
    ELSE
       I1 = FP_INTRSCT_RP(SFS_#,SRS_#,Y);
    IF HAVE_ALREADY_MADE_INSERTION & I1 ¬= 0 THEN
       DO;  CALL REPLACE(SP,I1);  RESTART;  END;
    IF HAVE_ALREADY_MADE_INSERTION THEN /*  DO SOMETHING  */
       DO;  CALL REPLACE(SP,SYM_LIST(READ_START(SFS_#)));  RESTART;  END;
    CALL STEPPING;
    /*  CHECK INTERSECTION OF APPROP. SETS  */
    IF R_STATE_TYPE = 0 THEN
       I = FP_INTRSCT_RP(SFS_#,SRS_#,W);
    ELSE I = FP_INTRSCT_RP(SFS_#,SRS_#,Y);
    IF I = 0 & I1 ¬= 0 THEN
         DO;  CALL REPLACE(SP,I1);  RESTART;  END;
    IF I = 0 THEN   /*  DO SOMETHING  */
       I = SYM_LIST(READ_START(SFS_#));
    INSERT_I_BEFORE_STACK_TOP;
    HAVE_ALREADY_MADE_INSERTION = TRUE;
    RESTART;
    /*  FP IN LA STATE...  */
H1:  IF CAN_DO_WITHOUT_TOP(X) = TRUE THEN RESTART;
    IF R_S_TYPE = 0 THEN
       I1 = FP_INTRSCT_RP(SFS_#,SRS_#,X);
    ELSE
       I1 = FP_INTRSCT_RP(SFS_#,SRS_#,Z);
    IF I1 ¬= 0 & HAVE_ALREADY_MADE_INSERTION THEN
       DO;  CALL REPLACE(SP,I1);  RESTART;  END;
    IF HAVE_ALREADY_MADE_INSERTION THEN  /*  DO SOMETHING  */
       DO;  CALL REPLACE(SP,MEMBER_FP_LA);  RESTART;  END;
    CALL STEPPING;
    IF R_STATE_TYPE = 0 THEN
       I = FP_INTRSCT_RP(SFS_#,SRS_#,X);
    ELSE I = FP_INTRSCT_RP(SFS_#,SRS_#,Z);
    IF I = 0 & I1 ¬= 0 THEN
       DO;  CALL REPLACE(SP,I1);  RESTART;  END;
    IF I = 0 THEN  /*  DO SOMETHING  */  I = MEMBER_FP_LA;
    INSERT_I_BEFORE_TOKEN;
    HAVE_ALREADY_MADE_INSERTION = TRUE;
```

```
   RESTART;
   /*  ALL CONDITIONS POSSIBLE AT THIS POINT (PTRS MATCHED):
       FP: READ, E          RP: READ, E
           READ, E              LA, E
           LA, E                READ, E
           LA, E                LA, E                          */
H2:  IF S_TYPE = 0 THEN DO;  W = 0;  Y = 2;  END;
     ELSE DO;  W = 1;  Y = 3;  END;
   IF CAN_DO_WITHOUT_TOP(W) = TRUE THEN RESTART;
   IF R_S_TYPE = 0 THEN  /*  RP ERROR IN READ STATE  */
      I = FP_INTRSCT_RP(SFS_#,SRS_#,W);
   ELSE  /*  RP ERROR IN LA STATE  */
      I = FP_INTRSCT_RP(SFS_#,SRS_#,Y);
   IF I = 0 THEN  /* DO SOMETHING  */
      DO;  IF S_TYPE = 0 THEN I = SYM_LIST(READ_START(SFS_#));
      ELSE I = MEMBER_FP_LA;
      END;
   CALL REPLACE(SP,I);
   HAVE_ALREADY_MADE_INSERTION = TRUE;
   RESTART;
   /*  PTRS ¬ MATCH, RP IN ERROR...  */
H3:  IF R_SP > SP + 1 THEN
      DO;  /*  SECOND ERROR LOCATED...  */
      OUTPUT = '';
      OUTPUT = '*** SECOND ERROR LOCATED WHILE REVERSE PARSING. ' ||
               ' ILLEGAL SYMBOL PAIR: '||CK_SYM(R_SP+1)||' '||CK_SYM(R_SP);
      RETURN FALSE;
      END;
   IF S_TYPE = 0 & R_S_TYPE = 3 THEN
      DO;  IF  CHECK_CONTEXT_OF_TOP_AND_TOKEN(W,Z,Y,W) THEN RESTART;
      ELSE DO;  CALL REPLACE(SP,SYM_LIST(READ_START(SFS_#)));  RESTART;  END;
      END;
   IF S_TYPE = 3 & R_S_TYPE = 3 THEN
      RETURN  CHECK_CONTEXT_OF_TOP_AND_TOKEN(X,Z,Z,X);
   IF S_TYPE = 3 & R_S_TYPE = 0 THEN
      RETURN  CHECK_CONTEXT_OF_TOP_AND_TOKEN(X,Y,X,X);
   /*  FP & RP IN READ STATES, BOTH IN ERROR...  */
   IF  CHECK_CONTEXT_OF_TOP_AND_TOKEN(W,Y,W,W) THEN RESTART;
   ELSE DO;  CALL REPLACE(SP,SYM_LIST(READ_START(SFS_#)));  RESTART;  END;
END;  /*  REVERSE_PARSING_SUCCESSFUL  */

                    /************************
                     *                      *
                     *    FORWARD PARSER    *
                     *                      *
                     ************************/
```

```
DECLARE CYCLE_CNT FIXED INITIAL(0);
DO FOREVER;   CYCLE_CNT = CYCLE_CNT + 1;
DO CASE(STATE_TYPE);

   DO;  /* READ VIA LINEAR SEARCH.  */
   IF CONTROL(BYTE('P')) THEN
   OUTPUT = '                ' || V(SYM_BEFORE_READ("FF" & STATE_STACK(SP)));
   IF RESET_2 THEN
      DO;  SP = SP + 1;
      IF SP = SAVE_SP THEN
         DO;  RESET_2 = FALSE;  NEXT_SYMBOL(SP) = DUMMY_TOKEN;  END;
      END;
   ELSE IF RESET_1 THEN
      DO;  SP = SP + 1;  RESET_1 = FALSE;
      NEXT_SYMBOL(SP) = DUMMY_TOKEN;  END;
   ELSE IF R_P_S THEN
      DO;  SP = SP2;
      NEXT_SYMBOL(SP) = NEXT_SYMBOL(SP2);
      VAR(SP) = VAR(SP2);  FIXV(SP) = FIXV(SP2);
      SP2 = SP2 + 1;
      IF SP2 > SAVE_SP THEN  /* HAVE CAUGHT UP, TURN OFF THIS CONDITIONAL  */
         DO;
         R_P_S, HAVE_ALREADY_MADE_INSERTION = FALSE;
         END;
      END;
   ELSE DO;  CALL PUSH_AND_READ;  #_FIXITS = 0;  END;
   /*  A SPECIAL CHECK FOR MISSPELLED "END"...  */
   IF STATE_# = 36 & NEXT_SYMBOL(SP) = IDENT THEN
      DO;  I = CHECK_RES_WORD_SPELLING(VAR(SP));
      IF I > 0 THEN  /*  FOUND MISSPELLED "END"  */
         DO;  NEXT_SYMBOL(SP) = I;  VAR(SP) = V(I);  FIXV(SP) = 0;  END;
      END;
   I = READ_START(STATE_#);
   J = I + RD_#(STATE_#);
   DO WHILE NEXT_SYMBOL(SP) ¬= SYM_LIST(I) & I < J;
      I = I + 1;
   END;
      /*  A SPECIAL CHECK FOR MISSPELLS FOLLOWING 'BEGIN'...  */
   IF NEXT_SYMBOL(SP) = BEGINV THEN
      DO;
      IF RESET_2 & DUMMY_TOKEN = IDENT THEN
         DO;  K = CHECK_RES_WORD_SPELLING(VAR(SP+1));
         IF K > 0 THEN
            DO;  DUMMY_TOKEN = K;  VAR(SP+1) = V(K);  END;
         END;
      ELSE  IF TOKEN = IDENT THEN
         DO;  K = CHECK_RES_WORD_SPELLING(BCD);
```

```
      IF K > 0 THEN
         DO;  TOKEN = K;  BCD = V(K);  END;
      END;
   END;
IF SHR(STATE_LIST(I),8) = 6 & SHR(STATE_BEFORE_READ,8) = 3 THEN
   STATE_STACK(SP), STATE_# = STATE_LIST(I);
 ELSE DO;  SFS_#,STATE_BEFORE_READ = STATE_STACK(SP -1);
      STATE_STACK(SP), STATE_# = STATE_LIST(I);
      END;
END  /* CASE 0  */;

DO;  /* READ VIA AN ARRAY ACCESS.  */
/*  ALGOL-E ¬ READ VIA THIS CASE  */
END;

DO;  /* REDUCE. */
MP = SP - #_TO_POP(STATE_#);
CALL SYNTHESIZE(STATE_#);
SP = MP;
IF R_P_S & #_TO_POP(STATE_#) > 0 THEN
   DO;  /*  CLOSE UP THE STACK  */
   K = SP + 1;  L = SP2;  SP2 = K;
   DO WHILE L ¬> SAVE_SP;
      NEXT_SYMBOL(K) = NEXT_SYMBOL(L);
      VAR(K) = VAR(L);  FIXV(K) = FIXV(L);
      K = K + 1;  L = L + 1;
      END;
   SAVE_SP = K - 1;
   END;
NEXT_STATE = REDUCE_SUCC(STATE_#);
END;

DO;  /* LOOK AHEAD (ORDINARY). */
DECLARE (LA_SYMBOL, BYTE_#,SHR_#) BIT(8);
IF RESET_1 | RESET_2 THEN LA_SYMBOL = DUMMY_TOKEN;
ELSE IF R_P_S THEN LA_SYMBOL = NEXT_SYMBOL(SP2);
ELSE LA_SYMBOL = TOKEN;
/*  NEED LOCAL FIXIT HERE AS ALGOL-E AIN'T REALLY
 SLR(1) BECAUSE OF THE <IF STATEMENT> CONSTRUCTION, IE. "ELSE"
 GUMS IT UP.
 NOTE:  CONFLICT ALSO EXISTS FOR THE DOUBLE SYMBOL ":="; THIS
 HAS BEEN FIXED BY USING A FALSE TERM. SYMBOL "<SETQ>", REF
 INITIALIZATION AND SCAN PROCEDURES.                          */
IF STATE_# = 2 & LA_SYMBOL = ELSEV THEN
   DO;  NEXT_STATE = FAIL_STATE(STATE_#); GO TO QUIT;  END;
/*  IF STILL WORKING ON DECLARATION SETS THEN NEED SPECIAL
    CHECK FOR THE TYPE DECLARATION RESERVED WORDS   */
IF LA_SYMBOL = IDENT & SYM_BEFORE_READ("FF" & STATE_STACK(SP-1)) = 56
```

```
   THEN DO;                          /*  <DECLARATION SET>  */
   K = CHECK_RES_WORD_SPELLING(BCD);
   IF K > 0 THEN
      DO;  TOKEN, LA_SYMBOL = K;  BCD = V(K);  END;
   END;
BYTE_# = SHR(LA_SYMBOL,3);
SHR_# = 7 - (LA_SYMBOL & "(1)111");
IF SHR(BYTE(LA_TABLE(STATE_#), BYTE_#), SHR_#)
THEN DO; NEXT_STATE = SUCC_STATE(STATE_#);   END;
ELSE DO;  NEXT_STATE = FAIL_STATE(STATE_#);  END;
QUIT: ;
END;

DO;  /* LOOK AHEAD (FOR A PRODUCTION WITH AN EMPTY RIGHT PART).  */
END;

DO;  /* LOOK BACK (AT ONE OR MORE EXCEPTIONS).  */
DECLARE PREV_STATE BIT(16);
PREV_STATE = STATE_STACK(SP - 1);
I = LB_START(STATE_#);
J = I + LB_#(STATE_#);
DO WHILE PREV_STATE ¬= LB_STATE(I) & I < J;
   I = I + 1;
END;
NEXT_STATE = RESUME_STATE(I);
END;

DO ;  /* ERROR. */
DECLARE PREV_ERR_CYCLE BIT(32) INITIAL("FFFFFFFF");
IF ¬R_P_S THEN DO;  SAVE_SP = SP;  SAVE_CARD_# = CARD_COUNT;  END;
IF NEXT_SYMBOL(SP - 1) = IDENT THEN
   DO;  /*  CASE OF <ID> 2 STATES BACK; CHECK FOR MISSPELLED
            RESERVE WORD ...  */
   I = CHECK_RES_WORD_SPELLING(VAR(SP-1));
   IF I > 0 THEN  /*  FOUND IT IN RESERVE WORD LIST  */
      DO;  RESET_2 = TRUE;  TSP = SP - 1;
      NEXT_SYMBOL(TSP) = I;  VAR(TSP) = V(I);  FIXV(TSP) = 0;
      DUMMY_TOKEN = NEXT_SYMBOL(SP);
      SP = SP - 2;
      STATE_# = STATE_STACK(SP);
      GO TO FIXED_IT;
      END;
   /* ELSE ASSUME THIS <IDENT> IS OK */
   END;
IF NEXT_SYMBOL(SP) = IDENT THEN
   DO;  /*  ...OR THIS <IDENT> MAY BE MISSPELLED */
   I = CHECK_RES_WORD_SPELLING(VAR(SP));
   IF I > 0 THEN
```

```
      DO;
      VAR(SP) = V(I);  FIXV(SP) = 0;  DUMMY_TOKEN = I;
      SP = SP - 1;  RESET_1 = TRUE;
      IF SHR(STATE_BEFORE_READ,8) = 3 THEN  /* LOOK AHEAD STATE  */
         STATE_STACK(SP), STATE_# = STATE_BEFORE_READ;
      ELSE
         STATE_# = STATE_STACK(SP);
      GO TO FIXED_IT;
      END;  /* IF I > 0 ...    */
   END;  /* IF NEXT_SYMBOL(SP) = ...    */
/*  ERROR NOT DETECTED AS SPELLING ERROR, SO TRY...  */
IF ERROR_ANALYZER THEN
   DO;
   R_P_S = TRUE;  #_FIXITS = #_FIXITS + 1;
   IF I_B4_ST | I_B4_TOKEN THEN  /*  BUILD NEW SYMBOL STACK  */
      DO;  SAVE_SP,K = SAVE_SP + 1;  L = K - 1;
      I_B4_ST, I_B4_TOKEN = FALSE;
      DO WHILE L ¬< SP;
         NEXT_SYMBOL(K) = NEXT_SYMBOL(L);
         VAR(K) = VAR(L);  FIXV(K) = FIXV(L);
         K = K - 1;  L = L - 1;
         END;
      NEXT_SYMBOL(SP) = I_SYM;  VAR(SP) = V(I_SYM);
      FIXV(SP) = 0;
      OUTPUT = '';
      OUTPUT = '** CORRECTION ** INSERTING "' || V(I_SYM) || '" BETWEEN "'
               || CK_SYM(L) || '" AND "' || CK_SYM(K+1) ||
               '" ON LINE ' || SAVE_CARD_#;
      END;
   /*  NOW CHECK FOR SYMBOLS TO BE DELETED...  */
   IF DELETING THEN
      DO;  SP1, SP2 = SP;
      DO WHILE SP1 ¬> SAVE_SP;
         IF NEXT_SYMBOL(SP1) = 255 THEN SP1 = SP1 + 1;
         ELSE DO;  NEXT_SYMBOL(SP2) = NEXT_SYMBOL(SP1);
            VAR(SP2) = VAR(SP1);  FIXV(SP2) = FIXV(SP1);
            SP1 = SP1 + 1;  SP2 = SP2 + 1;
            END;
      END;  /*  DO WHILE...  */
      DELETING = FALSE;
      SAVE_SP = SP2 - 1;
      END;
   SP2 = SP;  /*  WILL NEED THIS MARKER IN REDUCE STATE  */
   SP = SP - 1;  /*  SET SP IN PREPARATION FOR RESTARTING PARSING  */
   IF S_TYPE = 3 THEN  /*  LA STATE  */
      STATE_STACK(SP), STATE_# = STATE_BEFORE_READ;
   ELSE  /* READ STATE  */
      STATE_# = STATE_STACK(SP);
```

```
      IF #_FIXITS > 4 THEN
         DO;  #_FIXITS = 0;   GO TO FORGET_IT; END;
      GO TO FIXED_IT;
   END;   /* IF ERROR_ANALYZER...  */
   /*  REVERSE PARSING NOT SUCCESSFUL SO...  */
   FORGET_IT:  CYCLE_CNT = CYCLE_CNT - 2;
   HAVE_ALREADY_MADE_INSERTION = FALSE;
   IF PREV_ERR_CYCLE ¬= CYCLE_CNT THEN
   DO;
      PREV_ERR_CYCLE = CYCLE_CNT;
      PREV_STATE = STATE_STACK(SP - 1);
      OUTPUT = '';
      IF PREV_STATE <= "1FF"   /* IS IT A READ STATE?  */
      THEN P_SYM = V(SYM_BEFORE_READ("FF" & PREV_STATE));
      ELSE P_SYM = V(SYM_BEFORE_LA(STATE_#));
      IF STATE_# = 255 THEN N_SYM = V(NEXT_SYMBOL(SP));
                       ELSE N_SYM = V(LA_SYMBOL);
      OUTPUT = 'THE FORWARD PARSING ERROR IS:';
      CALL ERROR('ILLEGAL SYMBOL PAIR:  ' || P_SYM || ' ' || N_SYM);

      /*  PRINT THE CONTENTS OF THE PARSE STACK.  */
      MSG = 'PARTIAL PARSE SO FAR IS:  "';
      CALL TRACER(2, SP-1, MSG);
   END;
   OUTPUT = '';
OUTPUT = '** SORRY ABOUT THAT, BUT I CAN''T FIX IT...GOOD LUCK.';
   /*  GET RID OF THE JUNK ALREADY ON THE STACK.  POP BACK
       TO FIRST PERIOD, SEMICOLON, OR <BEGIN>...  */
   DO K := 0 TO SP - 1;
      I = SP - 1 - K;   /* SEARCH FROM TOP TO BOTTOM  */
      J = NEXT_SYMBOL(I);
         IF J=2 /*PERIOD*/ | J=5 /*SEMICOLON*/ | J=55 /*<BEGIN>*/ THEN
            DO;  SP = I;  GO TO GOTCHA;  END;
      END;
   GOTCHA:  IF NEXT_SYMBOL(SAVE_SP) = EOFILE
   THEN
   DO;
      OUTPUT = 'PROGRAM ENDS PREMATURELY.  COMPILATION TERMINATED.';
      GO TO END_OF_ALL_OF_IT;
   END;
   IF NEXT_SYMBOL(SAVE_SP) = SEMI THEN STATE_STACK(SP),STATE_# = 36;
   ELSE  /*  NEXT SYMBOL IS A PERIOD  */
   DO;
      IF TOKEN = IDENT THEN
         DO;  I = CHECK_RES_WORD_SPELLING(BCD);
         IF I > 0 THEN  /*  TOKEN = RES WORD  */
            DO;  TOKEN = I;  BCD = V(I);  END;
         END;
```

```
      IF TOKEN = 7 | TOKEN=31 | TOKEN=33 | TOKEN=34 | TOKEN=35 THEN
         DO;  STATE_STACK(SP),STATE_# = 67;  GO TO THATS_IT;  END;
      STATE_STACK(SP),STATE_# = 70;
      THATS_IT:  ;
      END;
   RESET_1, RESET_2, R_P_S = FALSE;
   OUTPUT = SUBSTR(POINTER,TEXT_LIMIT - CP + MARGIN_CHOP);
   OUTPUT = '** RESUME **';
   FIXED_IT:  RP_ERROR = FALSE;  /* SET NEUTRAL FLAG  */
   END  /*  OF ERROR RECOVERY CASE.  */;


   DO;  /*  EXIT.  */
      /*  RESET TO STARTING CONFIGURATION.  */
   SP, STATE_STACK(0), STATE_# = 0;
   TOKEN = 1;     /*  V(1) = '_|_'  */
   CYCLE_CNT = 0;
   RETURN;
   END;

END  /*  OF CASE(STATE_TYPE)  */;
END;
END SYNTACTICAL_ANALYZER;
```

LIST OF REFERENCES

1. DeRemer, F.L., Practical Translators for LR(k) Languages, Ph.D. Thesis, MIT, Cambridge, 1969.

2. DeRemer, F.L., "Simple LR(k) Grammars," Communications of the ACM, v. 14, no. 7, pp. 453-460, July 1971.

3. Gries, D., Compiler Construction for Digital Computers, Wiley, 1971.

4. Horning, J.J. and Lalonde, W.R., "Empirical Comparison of LR(k) and Precedence Parsers," SIGPLAN Notices, v. 5, no. 11, pp. 10-24, November 1970.

5. Irons, E.T., "An Error-Correcting Parse Algorithm," Communications of the ACM, v. 6, no. 11, November 1963.

6. Kildall, G., ALGOL-E: An Experimental Approach to the Study of Programming Languages, Internal Publication, Naval Postgraduate School, Monterey, 1972.

7. Knuth, D.E., "On the Translation of Languages from Left to Right," Information and Control, v. 8, pp. 607-639, October 1965.

8. LaFrance, J., "Optimization of Error Recovery in Syntax-Directed Parsing Algorithms," SIGPLAN Notices, v. 5, no. 12, pp. 2-17, December 1970.

9. Leinius, R.P., Error Detection and Recovery for Syntax Directed Compiler Systems, Ph.D. Thesis, University of Wisconsin, Madison, 1970.

10. McKeeman, W. M., Horning, J.J. and Wortman, D.B., A Compiler Generator Implemented for the IBM System 360, Prentice Hall, 1970.

11. Rich, L.V., Error Detection, Analysis and Recovery in XPL Based Compilers, Master Thesis, Naval Postgraduate School, Monterey, 1971.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Documentation Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0212 Naval Postgraduate School Monterey, California 93940	2
3.	LT(j.g.) Robert C. Bolles, USNR Code 53 Bq Department of Mathematics Naval Postgraduate School Monterey, California 93940	1
4.	LCDR Gordon T. McGruther, USN 18035 West Outer Drive Dearborn, Michigan 48128	1

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author)
Naval Postgraduate School
Monterey, California 93940

2a. REPORT SECURITY CLASSIFICATION
Unclassified

2b. GROUP

3. REPORT TITLE
An Approach to Automating Syntax Error Detection Recovery, and Correction for LR(k) Grammars

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)
Master's Thesis; (June 1972)

5. AUTHOR(S) (First name, middle initial, last name)
Gordon T. McGruther

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
June 1972	74	11

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
b. PROJECT NO.	
c.	9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)
d.	

10. DISTRIBUTION STATEMENT
Approved for public release; distribution unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
	Naval Postgraduate School Monterey, California 93940

13. ABSTRACT

An automatic, language-independent syntax error detection, recovery, and correction system for LR(k) grammars is proposed. The requirement is made that the reverse of the grammar involved is also LR(k). The implications and justification for this requirement are discussed. Given that the grammar is both LR(k) and RL(k), forward and reverse parsers localize errors and define left and right error context providing a strong base from which error analysis may proceed. Possible deterministic and heuristic corrective actions to follow error analysis are presented. The definition and selection of keys from the set of terminal symbols for the grammar which enable the reverse parser to be engaged upon error detection are discussed.

A model of the proposed system, implemented in an XPL compiler for a large ALGOL-like grammar, is described and the results of test programs are exampled and discussed.

Possible extensions to the system are presented and areas requiring further analysis are defined.

DD FORM 1 NOV 65 1473 (PAGE 1)
S/N 0101-807-6811

Security Classification

A-31408

KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
LR(k), RL(k)						
Reverse parsing						
Deterministic and heuristic error correction						
Keys						